图灵程序设计丛书

MANNING



DSLs in Action

领域专用语言实战

[美] Debasish Ghosh 著

郭晓刚 译

著名博客“Ruminations of a Programmer”作者

ACM高级会员20余年经验总结

多位业内大牛鼎力推荐

全面涵盖5种JVM语言

真正讲透DSL设计与实现



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明



作者简介

Debasish Ghosh Anshinsoft公司（http://www.anshinsoft.com）首席技术布道师，开发者推荐博客“Ruminations of a Programmer”的作者，ACM协会高级会员。他拥有跨国IT企业20余年工作经验，擅长为各种客户（无论是中小型公司还是财富500强企业）交付企业级解决方案，对自己将软件设计和编程最佳实践制度化而引以为傲，热爱Java、Ruby、Scala、OO及函数式编程，关注DSL和NoSQL数据库。电子邮件：dghosh@acm.org。Twitter账号：@debasishg。



译者简介

郭晓刚 大学肄业，有过两次创业和创业失败的经验，从嵌入式硬件到企业软件开发皆无所成。作为爱好的翻译反倒坚持不辍，积攒了五六本独译、合译的作品。长期在InfoQ中文站从事编辑工作，顺带磨练了技术触觉和翻译、写作的技艺。现在家照顾本书拖稿期间出生的儿子。

TURING 图灵程序设计丛书



DSLs in Action

领域专用语言实战

[美] Debasish Ghosh 著
郭晓刚 译

人民邮电出版社
北京

图书在版编目（C I P）数据

领域专用语言实战 / (美) 戈施 (Ghosh,D.) 著 ; 郭晓刚译. -- 北京 : 人民邮电出版社, 2013.11
（图灵程序设计丛书）
书名原文: DSLs in action
ISBN 978-7-115-33174-8

Ⅰ. ①领… Ⅱ. ①戈… ②郭… Ⅲ. ①程序语言 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第234324号

内 容 提 要

DSL（领域专用语言）的要旨在于沟通。精心设计的 DSL 可以以一种从外观到内在都极为自然的方式，传达出其所表示领域的本质和真意，帮助消除业务与技术的隔阂，促进项目干系人与程序员的沟通。

《领域专用语言实战》不仅介绍如何使用 DSL 解决问题，还会使用 Ruby、Groovy、Scala、Clojure 等现代语言阐述 DSL 的设计与实现，针对这些语言所代表的不同编程范式深入讨论其在 DSL 设计上的优劣。本书共分三部分。第一部分定位 DSL 驱动开发环境，寻找其在应用程序架构中的用武之地，帮助程序员或架构师了解如何调整现有开发工具和技术，使之适应 DSL 驱动的新范式。第二部分带你设计优秀的语义模型，使之成为上层语言抽象的有力后盾。该部分主要指导开发人员按照优秀抽象的设计原则搭建领域模型，由浅入深讲解了 DSL 实现技术，如元编程、解析器组合子，以及 ANTLR、Xtext 等开发框架。第三部分主要展望未来趋势，重点讨论解析器组合子和 DSL 工作台技术的发展前景。

本书适合开发人员、架构师、领域用户学习参考。

◆ 著　　　　[美] Debasish Ghosh
　序　　　　Jonas Bonér
　译　　　　郭晓刚
　责任编辑　毛倩倩
　执行编辑　李　静　张　庆
　责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
　邮编　100061　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　　印刷

◆ 开本：800×1000　1/16
　印张：19.5
　字数：479千字　　　　2013年 11 月第 1 版
　印数：1－4 000册　　　2013年 11 月北京第 1 次印刷

著作权合同登记号　图字：01-2011-2958号

定价：69.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

广告经营许可证：京崇工商广字第 0021 号

版 权 声 明

献给我的祖父，是他教会了我第一个字母。

序

我很喜欢摆弄编译器，只要是亲手打造自己的语言，无论动手还是动脑都是一种享受。编程语言，尤其是DSL（Domain Specific Language，领域专用语言），能激起我极大的热情。

DSL这一概念并不是最近才发明的，Lisp开发者们早就在开发和使用所谓的“小语言”了。不过近年来，DSL确实在整个行业范围内被更广泛地使用和接受，相关工具和技术也日渐成熟。如果开发者想探索语言设计这一精彩世界，可以说现在的条件是前所未有的。

如同大多数语言，DSL的要旨在于沟通。精心设计的DSL可以以一种从外观到内在都极为自然的方式，传达出其所表示领域的本质和真意。DSL能帮助消除业务与技术的隔阂，促进项目干系人与程序员的沟通。这种能力比以往任何时候都更重要，更值得我们去追求。

Debasish在Scala和开源社区里都是受人尊敬的专家。他的博客既给人以学识上的启发，又充满阅读乐趣，多年来我一直在关注。Debasish一年前开始为Akka项目贡献力量，我这个长年的读者因而与他有了进一步的接触。往来言行一下子就表露出来，他不仅是一位深刻的思考者，还是一位有行动力的实干家。从那以后，与他讨论编程语言、设计成了我乐在其中的爱好。

这是一本令人激动的书。书中内容的涵盖面很广，而在此基础上又有相当的深度，除了带领读者穿梭于DSL发展的最前沿，它还将带领大家思考如何设计灵活而自然的DSL。此外，读者还将领略Scala、Groovy、Clojure、Ruby等各具特色的语言，掌握每一种语言解决问题的思路和手段。开卷有益，诚哉斯言。

Jonas Bonér
Akka项目、AOP框架AspectWerkz创建人
http://jonasboner.com

前　言

2001年春天，我供职的Anshinsoft公司（http://www.anshinsoft.com）开始涉足企业应用开发业务，客户是一家在亚太区数一数二的证券中介和资产管理企业。这段经历激起了我对一个专门的问题领域进行建模，然后将模型转换成软件的兴趣。于是我开始了一段考验毅力的学习旅程，仔细参详了Eric Evans的领域驱动设计著作（*Domain-Driven Design: Tackling Complexity in the Heart of Software*[①]），聆听了Josh Bloch关于如何设计优秀API的教诲（How to Design a Good API & Why it Matters，http://www.infoq.com/presentations/effective-api-design）以及Martin Fowler关于DSL的教义。

精心设计的DSL旨在向目标用户提供人性化的界面，而做到这一点的最佳途径是让编程模型使用领域专用语言来"说话"。我们一直以来总是把程序设计得像个"黑盒"，很少让业务人员得知其内部细节，这种做法可以休矣。经验告诉我，所有用户都希望查看一下你建模在代码里的业务规则，而不是白板上杂乱的框框和箭头。

嵌在代码里的规则要容易被用户理解，用户必须能看懂你使用的语言，这就是我从事十年领域建模的领悟。当规则可被理解的时候，DSL也就呼之欲出了。随之得到改善的不仅有开发团队和业务人员的沟通效率，还有软件面向用户的表达能力。

对于我们能否为用户提供表现力充沛的语法和语义，实现语言无论何时都是一个决定性的因素。有赖于当今生态环境的巨大发展，我们所设计的语言得以在表现力上有了长足进步。以鼓励开发者编写精炼而富有表现力的代码而论，Ruby、Groovy、Scala和Clojure是先行的表率。在这几种语言下的第一手编程经验让我感觉到，它们的语言风格和表达习惯远比大多数前代语言更适合领域建模。

写这样一本关于DSL的书是很大的挑战。我试图关注DSL的一切现实事物，所以从一开始就设定了具体的领域。当我们渐次展开论述，领域模型随着各种业务需求的累加而变得越来越复杂。这正好充分体现了DSL驱动的开发方式对问题域复杂度增长的适应能力。DSL方式并不是对API设计的颠覆，它只是鼓励你在API的设计思路上多考虑一个维度。请务必记住，你的用户才是DSL的使用者。凡事多从用户的角度去考虑，你一定会成功的！

① 中文版《领域驱动设计》，人民邮电出版社出版。——编者注

致　　谢

首先，这可能有点儿不同寻常，我想感谢一下自己。我完全没料到自己能把注意力集中在一个兴趣点上这么久。编写本书的过程中，我真正意识到了毅力、决心和信念的价值。

感谢Anshinsoft的同事们创造了能够培育想法并让想法腾飞的工作环境。夜半之时的头脑风暴帮助我雕琢出了许多复杂的领域模型，也引发了我对DSL的热爱。

感谢我们的客户代表渡边桑（Tohru Watanabe先生），我从他那里学到了证券交易业务的领域模型。这本书里满是他多年来教给我的例子。

感谢以下审阅者帮助我提高书稿的质量：Sivakumar Thyagarajan、Darren Neimke、Philipp K. Janert、James Hatheway、Kenneth DeLong、Edmon Begolli、Celso Gonzalez、Jason Jung、Andrew Cooke、Boris Lenzinger、David Dossot、Federico Tomassetti、Greg Donald、John S. Griffin、Sumit Pal、Rick Wagner。特别感谢审阅者Guillaume Laforge和John Wilson指正Groovy DSL的编写细节，感谢Michael Fogus对第5章和第6章内容的建言，感谢Sven Efftinge对第7章中Xtext和外部DSL的意见。我还要感谢Franco Lombardo在本书付印前的紧迫时间里对文稿的最后技术审读。

Twitter网友在本书的写作过程中贡献了许多真知灼见，给了我无可估量的帮助和启发。

Manning出版社有一支优秀的团队，感谢他们的实心协助。项目编辑Cynthia Kane在文法和写作风格上不知疲倦地给出了指点，还站在读者的角度与我探讨了本书每一章的内容。如果读者觉得本书文字简单易懂，那要归功于Cynthia一遍又一遍的审读，是她敦促我一遍又一遍地修改行文。感谢Karen Tegtmeyer组织同行评议，感谢Maureen Spencer在本书撰写的全过程给予的帮助，感谢Joan Celmer在编辑过程中的积极响应，感谢Manning出版社对本书制作提供了大力支持的所有工作人员。我还要感谢出版人Marjan Bace对我的信任。

向我的夫人Mou致以特别的谢意，她在我写书的日子里一直激励我。这段漫长而辛劳的旅程因为她不同时期的鼓舞而有了非凡的意义和累累硕果。

关 于 本 书

每一次我们在白板上设计领域模型，似乎总会在落实到代码的时候于纷杂中走了样。实现模型不管用哪种编程语言来表述，它都已经不是领域专家能理解的业务语言形式。白板上的模型是否精确反映了我们与领域用户商定的需求规格，这也无从让掌握领域规则的人员去验证。

对于这个症结，本书给出的解决之道是采用一种以DSL驱动的应用程序开发模型。假如我们围绕领域用户能够理解的语法和语义设计领域API，那么即使在应用程序代码的开发过程里，用户也能随时检查领域规则实现得是否正确。采用领域语言的代码更容易让人看懂，在这一点上，开发人员、维护人员、只懂业务不懂编程的领域专家都是受益者。

本书除了教你*使用*DSL来解决问题，还会教你*实现*DSL。在本书看来，DSL只是在语义模型外面包裹上薄薄一层以语言形态呈现的抽象。语义模型是把握领域核心结构的实现载体，语言层则使用领域用户的专门用语。

本书将使用Ruby、Groovy、Scala、Clojure等现代语言来讲授DSL的设计与实现，针对这些语言所代表的不同编程范式深入讨论它们在DSL设计上的长处和短处。读完本书，你将透彻理解一些必须掌握的概念，能够设计出用户理解且欣赏的优美的领域抽象。

读者对象

如果你希望自己设计的API其表现力既满足领域用户的需要，又能达到程序员同行的要求，那么这本书恰好适合你。如果你是一名领域用户，正期待着改善与开发团队的沟通效果，那么这本书恰好适合你。如果你是一名程序员，正为如何与领域用户核对业务规则的实现正确与否而苦恼，那么这本书同样适合你。

本书内容

图1、图2、图3除了勾画出了全书的组织脉络，对各章的内容也作了简略的阐述。本书分为三部分：

- 使用DSL；
- 实现DSL；
- DSL开发的未来趋势。

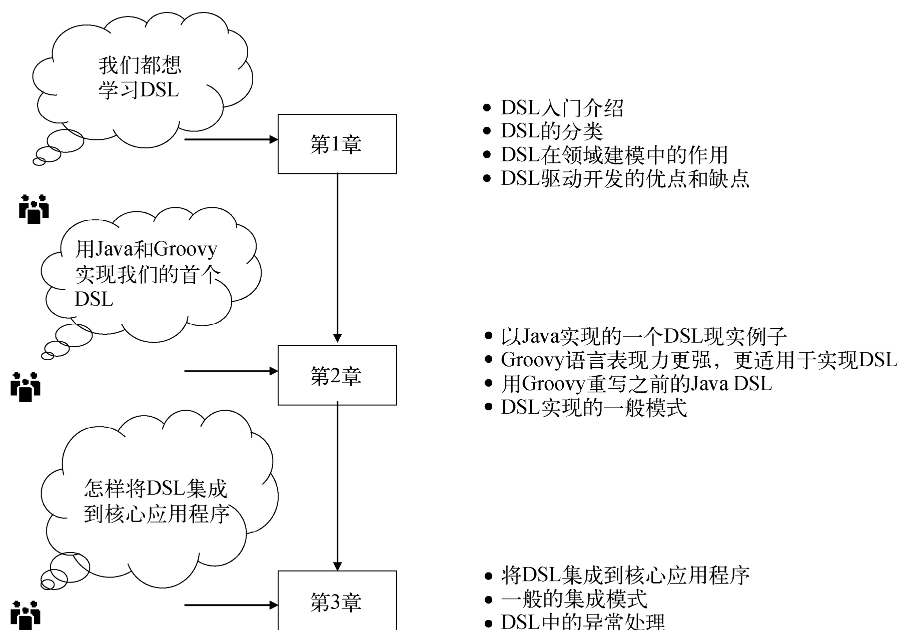


图1　第1章到第3章的学习历程

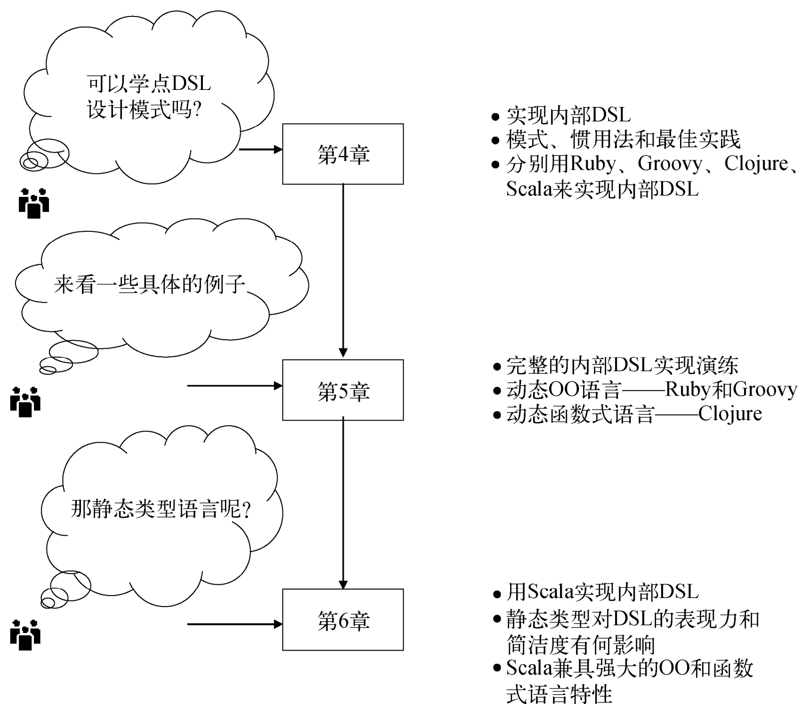


图2　第4章到第6章的学习历程

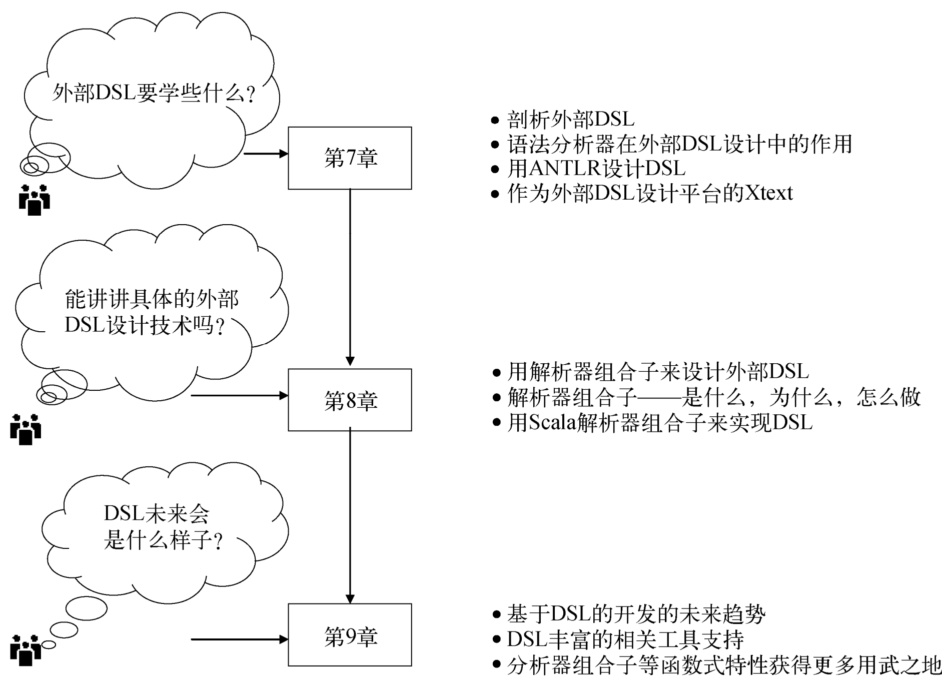


图3　第7章到第9章的学习历程

第一部分（第1章~第3章）作为总括，详细地阐述了DSL驱动开发环境的定位，帮助读者在自己的应用程序架构中找到它的用武之地。如果你是程序员或者架构师，这部分内容将协助你调整现有的开发工具和技术，使之适应DSL驱动的新范式。本书主要围绕各种JVM（Java虚拟机）语言展开论述。因此，Java程序员很快就能够从书中找到适合自身项目情况的DSL运用方式，在自己的Java项目内集成用表现力更佳的其他JVM语言开发出来的DSL。

DSL拥有各种贴近用户思维的语法结构，这些语言抽象有赖于语义模型在背后提供支撑。第二部分（第4章~第8章）探讨如何设计出优秀的语义模型，使之成为上层语言抽象的有力后盾。这个部分主要是给开发人员准备的，旨在指导开发人员按照优秀抽象的设计原则来搭建领域模型。从第4章到第8章，各章都含有丰富的DSL代码片段，实现语言包括Ruby、Groovy、Clojure、Scala等。如果正在或即将使用这些语言来实现DSL，那么你会发现这几章的内容特别实用。书中讲解了DSL的实现手法，而且将从最基本的技术入手，逐渐深入到高级技术，如元编程、解析器组合子，以及ANTLR、Xtext等开发框架。

第三部分（第9章）主要展望未来趋势，重点讨论解析器组合子和DSL工作台技术未来的发展。

本书面向真正的实践者。因此，虽然书中也含有理论知识，但只是作为帮助理解具体实现的铺垫而存在。我发自内心地相信这将是一本让奋战在开发第一线的实干家感觉有用的书。

排版约定

本书正文中穿插了不少插入内容和补充内容，用于提醒读者注意一些重要信息。

一般来说，下面的排版样式用于展示与证券交易及结算领域有关的信息。


金融中介系统

带有这个标志，即表示其中信息与DSL所属领域有关，读者需要了解其中知识才能理解上下文。此类内容一般是对一些特殊术语和概念的背景介绍。

书中还有带另一种标志的插入内容，其排版格式如下。


此类插入内容含有不属于所在章节讨论话题的知识。例如某种DSL设计的特殊惯用法、对前文讨论的重点归纳，或者我希望强调的其他重要知识点。

另外，我还用下面所示的标志来引起读者对特定内容的注意。


语言相关信息

看到这个标志，你就应该知道其中含有当前示例所用编程语言的小知识。你需要掌握这些特定的概念才能真正理解当前示例。

在这里，我提请大家注意不要轻易忽略这些带有不同标志的补充信息，它们都是可以帮助你透彻理解当前讨论内容的重要参考知识。

代码约定和下载

本书包含大量的DSL示例，其中不少例子的完成度很高，足以完整解释某一方面的领域规则实现。这些例子使用的编程语言有Java、Ruby、Groovy、Scala和Clojure。代码清单和正文中插入的代码片段都使用等宽字体，便于读者把它们和一般的文字区分开来。正文中出现的方法名、参数、对象属性、ANTLR和Xtext等脚本、XML元素和属性也都一律使用等宽字体呈现。

示例代码不一定总是短小的片段，有时候为了充分说明所涉领域的上下文和语义，也会占用较长的篇幅，保留较多的细节。书中的代码经常会为了适应书本的页面宽度，而在断行和缩进上做一些格式调整；偶尔还有调整不过来的情况，这时对于那些不得不折行的代码，我们会在折行的位置打上一个续行标记。

很多代码清单中会穿插一些标注，以便向读者提示重点。有时候标注还会带有数字编号，方便我们在后续的介绍中引用和参照。

书中用了多种编程语言来实现DSL，显然不可能所有的读者都熟悉其中所用的每一种语言。因此作为快速的参考，书后准备了每种语言的速查表格，见附录C到附录G。附录中的介绍只是

针对书中探讨DSL实现所用到的一些关键语言特性展开，并不完整全面，进一步的知识需要读者到表格后补充的参考资料中去寻找。

大多数时新的IDE都有能力支持开发者在同一项目中使用多种语言。如果读者不熟悉多语言开发环境，附录G是一个简单的入门指导。

书中所有示例的源代码都可以从Manning出版社网站下载[①]，地址为http://www.manning.com/DSLsinAction，配置构建环境和执行环境的相关指示也包含在内。阅读的时候在手边备一份源代码，这会对你很有帮助。

作者在线

本书有一个由Manning出版社运营的关联网络论坛，购买本书的读者具有免费访问论坛的权利。读者可以在上面发表评论、询问技术问题，并获得作者和其他用户的帮助。注册及使用论坛请访问http://www.manning.com/DSLsinAction。读者可从该页面了解论坛的注册和使用方法、论坛内提供的帮助、论坛守则等信息。

Manning出版社向读者承诺提供读者之间、读者与作者之间展开有意义对话的便利场所。作者只是志愿（且无偿）地参与论坛活动，因此Manning出版社不对作者参与论坛的程度做要求。我们建议读者尽量提出一些具有挑战性的问题，让作者有兴趣持续访问本论坛。在书籍在版期间，出版社网站将保证读者可以访问作者在线交流论坛及论坛上积累的讨论内容。

① 也可在图灵社区本书页面（http://www.ituring.com.cn/book/836）免费注册下载。——编者注

关于封面图片

本书封面图片为“来自克罗地亚斯拉沃尼亚地区奥西耶克城附近的久尔杰瓦茨村的男人”。这幅画是在克罗地亚历史名城斯普利特的民族博物馆一位热心馆员的帮助下取得的，来自该馆2003年重版的一本19世纪中期由Nikola Arsenovic编撰的克罗地亚传统服饰画集。斯普利特民族博物馆本身即坐落于中世纪的城市中心，也是罗马古迹的核心所在——建于公元304年前后的罗马帝国宫殿戴克里先宫遗址上。画集内收录了克罗地亚各地区人物形象的精细彩绘图样，并配有对服饰和生活习俗的文字说明。

久尔杰瓦茨村位于奥西耶克城附近，属于克罗地亚东部历史悠久的斯拉沃尼亚地区。斯拉沃尼亚的男人们传统上穿戴红色的帽子、白色衬衣、带刺绣图案的蓝色马甲和长裤，然后点缀上毛织或皮革的宽腰带和厚毛袜，最后罩一件棕色羊皮的短外套，也就是本书封面上的样子。

人们的衣着式样和生活习俗在最近的200年里发生了很大的变化。从前各地丰富多彩，极具地方特色的衣着和习俗已经消失殆尽。依现在的情况，甚至连不同洲的居民都趋于同化、难以区分了，相距数里的村落和市镇之间，就更不可能有什么差别了。也许，我们已经牺牲了文化的多样性来换取多姿多彩的个人生活——五光十色的、快节奏的科技生活。

Manning出版社特意从旧书和藏品里找回这些古老图样，让两个世纪以前丰富多彩的地方生活特色在图书封面上重焕光彩，以此来表达对计算机行业的创造精神和主动精神的赞美。

目　　录

Part 1 第一部分

领域专用语言入门

什么是领域专用语言（DSL）？ DSL 对于应用程序开发者有何价值？ DSL 能给使用软件的行业用户带来哪些好处？ DSL 驱动开发是否有助于开发团队与领域专家团队之间的交流？DSL 驱动开发有何利弊？这些问题都可以在第一部分找到答案。

这一部分由第 1 章 ~ 第 3 章组成，将介绍多种被广泛使用的 DSL 和设计 DSL 的一般原则，以便你在自己动手设计 DSL 时知道应该注意什么。

第 1 章照例是对 DSL 的入门介绍。

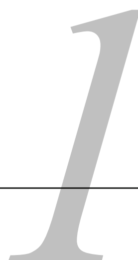
第 2 章将带你一起设计第一个 DSL。随着设计的推进，你将体会到用户需求一步步地演变成富于表现力的 DSL。我们首先用 Java 来实现 DSL，然后换成 JVM 上的另一种语言 Groovy，届时你将看到语言表现力的提升。

第 3 章介绍如何围绕一个核心应用程序集成内部与外部 DSL，以及如何处理错误和异常。

这一部分面向程序员和不具备编程背景的领域用户，因此特意避开了具体的实现细节，以便读者对 DSL 的大体情境有一个全面的了解。

第 1 章

初识DSL



本章内容

- ❑ 什么是DSL
- ❑ DSL对于商业用户和解决方案的开发者各有什么好处
- ❑ DSL的结构
- ❑ 采用设计得当的抽象概念

清晨上班路上，你通常都会走进钟爱的咖啡店点一杯“大杯纤体肉桂带奶油拿铁”，店员则会准确无误地端上一杯用脱脂奶和无糖糖浆调制的香甜肉桂口味473 ml拿铁咖啡，上浇打发的鲜奶油。因为你点单用的是她能理解的精确语汇，所以即使没有详细解释每个词的含义，也丝毫无碍于交流，哪怕不相干的人听了会摸不着头脑。本章将要介绍的就是如何用特定领域的语汇来表达一个问题，然后进一步在解答域对问题建模。这种从问题域映射到解答域的实现模型就是DSL（Domain-Specific Language，领域专用语言）的基本思路。如果把上述咖啡店里的情境做成软件，那么客人们每天点单所用的语言就是你要找的DSL。

开发者设计的任何应用程序都将问题域映射成解答域的实现模型。DSL是映射过程中的一项重要产物和组成部分。在更确切地定义DSL之前，我们首先介绍成功建立映射的必要过程。要使映射成立，你需要先找出两个领域之间相通的语汇。这组语汇是促成DSL最终诞生的关键种子。



设计得当的DSL实现必定不能缺少一套好的抽象。某些读者可能打算继续深究怎样设计出良好的抽象，因此我们在附录A中详细探讨了设计中应该追求的一些特质。你不妨现在就翻阅一下附录A，然后再继续看本章接下来的内容。1.7节也介绍了关于抽象的基本内容，但附录A的介绍要详细得多。

1.1　问题域与解答域

领域建模是帮助你分析、理解并识别某项具体活动所有参与方的活动。第一步从问题域入手，确定领域中的实体如何与其他实体进行有意义的互动。在咖啡店的例子里，你点单时用了该领域最自然的语言，用了与店员的知识最贴近的专门用语。术语构成了问题域的核心实体。咖啡店店员之所以能顺利给你提供相应饮料，正是因为你们俩都熟悉必要的专门用语。

1.1.1 问题域

在领域建模活动中，问题域指构成你所分析业务的那些过程、实体和约束条件。领域建模，也称领域分析（参见1.9节文献[1]），就是要识别出领域中所有的重要元素以及它们之间的协作关系。在前面的例子中，店员掌握了构成其问题域的所有实体，如咖啡、打发鲜奶油、肉桂、脱脂奶等。如果要分析一个更复杂的领域，比如金融中介的交易结算系统，那么证券、股票、债券、交易、结算就是其中的一些元素。除了这些，你还要研究证券如何发行、在交易所买卖、在各交易方之间结算、记录到各种账册和户头。你需要先确认这些协作关系，然后进行分析并把结果作为分析模型的产物记入文档。

1.1.2 解答域

问题域的分析模型是用解答域提供的工具和手段实现出来的。你只要点单，店员就懂得该如何制作相应咖啡。她所遵守的制作过程以及使用的工具就是其解答域的构成成分。面对的问题域越大，你可能就越需要从解答域寻求更多工具、方法学和技术手段方面的支持。问题域的元素需要映射成解答域中适当的技术手段。如果将面向对象方法作为解答域的基本平台，那么类、对象和方法就是解答域的基本组件。你可以把这些组件组合成大一点的组件，而后者可能正好能更好地表示问题域更高一层的元素。图1-1描绘了领域建模的第一步。如何从问题域出发，运用领域专家能理解的技术手段，完成向解答域转换的全过程？随着学习的不断推进，你对此过程的理解会逐渐加深。

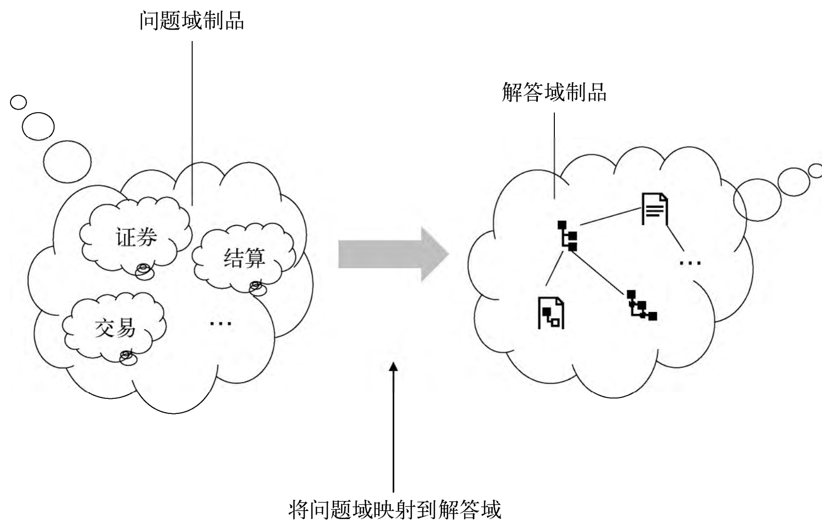


图1-1 问题域的实体和协作关系必须映射成解答域中相应的制品。图中左边的实体（证券、交易、结算等）需要在右边能找到对应的表示

领域建模的基本实践活动就是把问题域映射到解答域的若干制品，让所有的元素、相互作用和协作关系都得到正确而合理的表示。为此，你首先要把领域对象按合理的粒度归类。当分类正确时，

问题域的每个对象及其结构和语义都能在解答域找到对应项。不过，映射的效果无法超越作为两个领域之间互动媒介的语言的表现力。可靠的互动要求问题域与解答域分享一套共通的语汇。

1.2 领域建模：确立共通的语汇

要开始一项领域建模活动，首先要从等待被建模的问题域着手。你要理解域中实体彼此互动及履行各自职责的方式。这项工作要在领域专家和建模人员的协作之下进行。领域专家是内行，他们用领域内的专门用语交流，而且向外界解释领域概念的时候也离不开这套专门用语。建模者懂得如何将对模型的理解表达成一种可被编写成文档、分享并实现成软件的形式。建模者必须能理解领域专家的术语，并将其理解反映到设计的领域模型中去。

之前，我接手过一个对一家大型金融中介机构的后台操作完成建模的项目。我不是那个领域的专家，而且对证券业的业务活动涉及的种种细节和复杂情况所知甚少。经过与领域专家一段时间的合作，我感觉这个领域有足够的代表性，其例子和解释足以作为读者对其他领域建模时的参考。接下来的补充内容“金融中介系统：背景知识”介绍了证券交易和金融中介领域的基本情况，我们将它作为实例来介绍如何实现DSL。随着学习的深入，你将发现一些新的概念及必要的详细内容。如果你不熟悉股票交易也无需担心，我会用补充内容的形式提供足够的背景资料，帮助你了解建模对象的基本概念。

就在需求分析会议开始的第一天，金融业的领域专家开始大谈附息债券、折价债券、抵押、公司行为 。这些词都是金融中介的常用术语，但我完全不知道是什么意思。而且不少词其实是同义词，比如折价债券意思等同零息债券，不同的领域专家会在不同场合交替使用它们。可因为我不懂，混淆的情况层出不穷。在场的不可能都是金融业专家，所以我们很快意识到必须确定一套共通的语汇，以免知识交流会议失去意义。不仅与领域专家的协作要在共通语汇的前提下进行，而且我们要确保设计开发出来的模型基于同一种“语言”——这个领域的自然语言。



金融中介系统：背景知识

金融中介的业务始自一次交易过程。该过程涉及两个以上当事人之间证券与现金的交换，这些当事人称为交易方。在某个确定的日期（称为交易日），交易方承诺在股票交易所这个地点，按照商定的价格（称为单位价格）履行交易（成交）。交易过程的一大支柱——证券（另一支柱是现金）——有多种类型，如股票、债券、共同基金等，许多类型各自又有不同的分类体系。比如，债券又可分为附息债券和折价债券。

在交易日之后规定的天数内，基金或证券的所有权在交易方之间完成转移，这称为结算过程。每种证券都有各自的交易、成交、结算流程，在交易和结算过程中要经历一系列的状态变更。

共通语汇的益处

共通语汇在模型的所有关联方之中共享，作为一股维系力量把组成实现的各部分制品统一起来。更重要的是，有了这套共通语汇，你就可以在项目交付周期的各个阶段轻松跟踪各项特性、

功能和对象的变化轨迹。建模者用来编写测试用例的名词术语出现在程序里就是模块的名字，出现在数据模型里就是实体的名字，出现在测试用例里就是对象的名字。以这样的方式，共通语汇成为了架通问题域与解答域的桥梁。在项目前期，建立共通语汇花费的时间可能超过预期，但我几乎可以担保，它将帮助避免更多未来返工的时间。我们来看看共通语汇可以带来的切实的好处。

1. 把共通语汇当做粘合剂

在需求分析阶段，一套共通语汇可充当建模者和领域专家之间互相理解的桥梁，可使讨论更加简明扼要、效率更高。当交易员老鲍提到债券的应计利息时，建模员乔知道他说的债券特指附息债券。

2. 测试用例中的共通词汇

共通语汇还可作为开发测试用例的基础，这样便于领域专家验证测试用例的正确性。在我那个金融中介系统的项目中有这样一个测试用例：*对于证券公司蹦蹦高证券以40%价格发行、面值为10 000美元、首次计息日为2001年5月15日的零息债券，投资者应在发行时支付4000美元。*这个测试用例无论建模者、测试者还是负责审阅的领域专家都能完全理解，因为它采用了最自然的领域语言作为编写用语。

3. 开发中的共通语汇

如果开发团队用共通语汇来表述程序模块，那么产生的代码也将使用同一种领域语言。如果你提起模块的时候都用“债券交易模块”、“证券结算模块”一类的字眼，那么写代码的时候自然就会用同样的字眼命名领域实体。

在问题域与解答域之间发展出一套共通语汇是走向解答域的第一步。让我们给图1-1添上“共通语汇”这个维系两个领域的粘合剂，结果如图1-2所示。

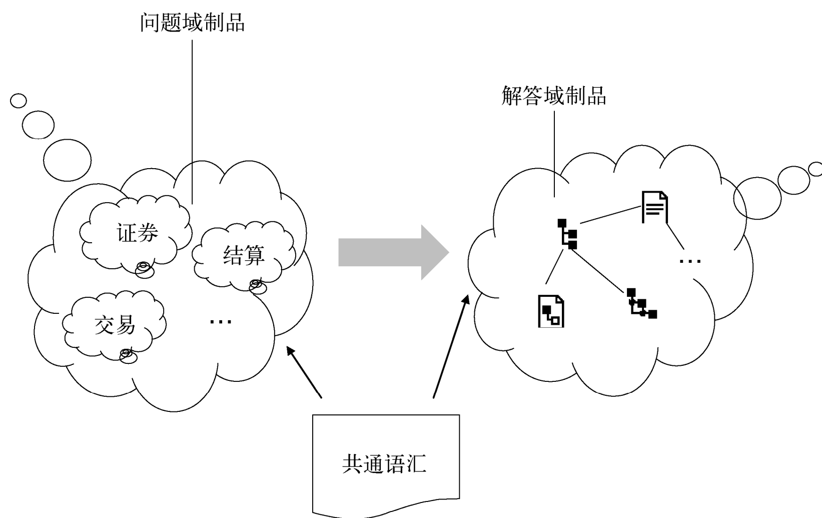


图1-2　问题域和解答域享有共同的语汇，可降低信息传达的困难度。在共通语汇之下，你可以从问题域的制品追踪到它在解答域的相应表示

你已经知道开发者和领域专家要有共通语汇，但是语言要怎样映射到解答域呢？对于开发者制作的模型，领域专家能理解多少？沟通问题是软件开发生态系统中的常见问题。

看过图1-2，你应该会知道不可能指望领域专家直接理解目前构成解答域的技术制品，他们不具备那样的能力。随着系统复杂度的提高，模型开始膨胀，沟通的障碍也越来越多。可是领域专家其实没必要理解那些与实现模型相关的复杂细节，验证业务规则实现得正不正确才是他们该做的事情。理想情况下，领域专家可以自己编写测试脚本去验证领域规则的实现的正确性及完善程度，但这不是一个现实的方案。

那么有没有可能以共通语汇为基础，为领域专家建立一种沟通模型，并让其他人也都能流利说出领域专家的日常业务用语？可以办到。这正是DSL发挥作用的时刻！

1.3　初窥 DSL

蹦蹦高证券公司的IT主管乔搞不懂交易员老鲍在做什么事情，不自觉瞥了一眼他的电脑屏幕。令他感到惊讶的是，乔发现老鲍正忙着往编程环境——乔觉得只有其手下的开发团队才能用的编程环境——里敲一些命令和语句。下面是他们之间的对话。

- 乔：嘿，老鲍，你还会编程？
- 老鲍：嗯，会点儿，可以在我们的新系统TrampolineEasyTrade中编点儿。
- 乔：可你不是交易员吗？
- 老鲍：交易员怎么了？交易员就用这个软件做交易。
- 乔：软件是提供给你们使用的，没打算让你们在里头编程。而且，这产品还没开发完呢。
- 老鲍：可既然是我将来要用的软件，现在我给它编一些测试不是挺好吗？这样一来，我的意见可以尽早反馈给开发团队；近距离参与，自我感觉贡献更大；我会对开发中的产品更有信心。而且，我还可以验证我的用例能不能通过。
- 乔：但这是开发团队的职责！我每天都和他们沟通。我们有工具检查代码覆盖率、测试覆盖率以及各种指标，肯定能保证交付最好的软件。
- 老鲍：就对金融中介系统这个领域的了解而言，你觉得谁更懂？我，还是你的那套工具？

最后，乔不得不认同老鲍身为金融中介系统领域的专家，更有能力检查新交易平台是否正确、充分地满足了功能规格书的要求。只不过乔还是不懂，为什么老鲍不是程序员却也会用测试框架写测试。

读者想必也有同样的疑惑。那就来看看下面的代码清单，这正是"刚刚"老鲍在屏幕上敲出的内容。

代码清单1-1　用DSL编写的交易单处理过程

```
place orders (
  new Order to buy(100 sharesOf "IBM")
    limitPrice 300
    allOrNone
    using premiumPricing,
  new Order to buy(200 sharesOf "CISCO")
```

```
      limitOnClosePrice 300
      using premiumPricing,
    new Order to buy(200 sharesOf "GOOGLE")
      limitOnOpenPrice 300
      using defaultPricing,
    new Order to sell(200 bondsOf "SUN")
      limitPrice 300
      allOrNone
      using {
        (qty, unit) => qty * unit - 500
      }
  )
```

好像真是代码呢。没错，但同时代码里的一些用语就跟老鲍平常坐上交易台时说的那种语言一样。老鲍正在编写一段创建新交易单的脚本，里面按照不同定价策略生成了各种交易单的样本。除了预定义的策略，他还可以在下单的时候自定义一种定价策略。

老鲍编程用的是什么语言？只要能完成工作，他一点儿都不在意。对于他来说，这种编程语言跟他在交易台上用的没什么两样。不过对于我们来说，老鲍所做的事情与程序员们日常编写代码的工作有何不同，这值得细辨一番。

- 老鲍的编程语言用语契合他所属的领域。他可以把平日在交易台前为客户下单所用的术语原封不动地写进测试脚本。
- 他所用的语言，从刚才所见的那一段来看，并不适用于金融中介业务以外的领域。
- 这种语言的表现力很强，老鲍只需要照着平常给客户创建新定单的步骤按部就班，就能把要做的事情清晰地表达出来。
- 这种语言语法简明。高级编程语言中常见的复杂语法细节奇迹般地不见了。

老鲍所用的正是一种为金融中介系统量身订做的*领域专用语言*。此刻，背后用什么语言来实现DSL显得无关紧要。我们很难从代码清单1-1中看出来背后用的是哪种语言，这正好说明设计者成功地为该领域创造了一种富于表现力的语言。

1.3.1 何为 DSL

DSL是一种针对特定问题的编程语言，我们平常所用的编程语言用途更为宽泛。DSL含有建模所需的语法和语义，在与问题域相同的抽象层次对概念建模。例如，你要点一份肉桂拿铁咖啡，就对店员使用她所掌握的领域语言。

定义 抽象是人类大脑的一种认知过程，它使我们集中注意力于认知对象的核心层面，忽略不必要的细节。1.7节会进一步讨论抽象与DSL设计的关系，附录A则完全是关于抽象的内容。

用DSL写出来的程序，任何一方面的品质都不应该低于用其他计算机语言编写的程序。DSL还应该赋予你设计领域中抽象概念的能力。在问题域可以用小的实体搭建出大的实体，那么在解答域，设计得当的DSL应该给予你同样灵活的组合能力，让你能够就像编排问题域的各种机能一样编排起各种DSL抽象。

现在你已了解什么是DSL，接下来看看它与你用过的其他编程语言有何不同。

1. DSL与通用编程语言的区别

领域专用语言这个名字其实已经给出了答案。你应该牢记DSL最重要的两个特征：

- 一种DSL专门针对一个特定的问题领域；
- DSL含有建模所需的语法和语义，在与问题域相同的抽象层次对概念建模。

用DSL编程时只需要处理问题域的复杂性，你用不着操心解答域的实现细节和其他非必要因素。（关于非本质复杂性的讨论，参见附录A。）因此，多数情况下非专业程序员也能用好DSL，前提是DSL具备了适当的抽象层次。数学家能轻松学会使用Mathematica进行工作，UI设计师写起HTML来怡然自得，就连硬件工程师都有VHDL（超高速集成电路硬件描述语言，是一种在电子设计自动化即EDA领域使用的DSL）可用，这些都是非专业程序员使用DSL的例子。因为要适应非程序员，DSL必须比通用编程语言更符合用户的直觉。

程序并不是一次写完了事，之后还要维护更新很多年，而其中负责“照料”程序的人很可能并没有参与设计最初的版本。因此，沟通是一个关键问题：程序要有能力与它的目标读者沟通。对于DSL，编译器和CPU都不是它的直接读者，有心理解程序行为的人类大脑才是它的“倾诉对象”。语言要利于交流，要让代码片段能够充分体现出建模者的思考过程。这就要求在设计DSL的时候为语法和语义都找准适合用户的抽象层次。

2. DSL对业务用户的益处

讨论到这里，我们可以总结出DSL区别于一般高级编程语言的两大特质，如下。

- DSL给予用户更高层次的抽象。也就是说用户不必分心于具体数据结构的微妙差别等低层次细节，而是专注于解决手头的问题。
- DSL只提供有限的语汇，不超出它的领域范围。正因为DSL排除了多余的东西，它能够帮助用户专注于被建模的问题。DSL的“视野”不似通用编程语言那般横向发散 。

对于不懂编程的领域专家，这两种特质使DSL成为了更便利的工具。业务分析员了解的领域，就是DSL抽象出来的领域。

随着越来越多的编程语言支持更高层次的抽象设计，各种DSL正翘首以待成为现今程序开发生态系统的重要组成部分。不懂编程的领域分析师肯定会在其中扮演重要的角色。如果有DSL，分析师从一开始就能撰写正确的测试脚本。测试脚本不是为了立即运行，而是用来保证编写程序时充分考虑到各种可能的业务场景。只要DSL的设计找准了抽象层次，让领域专家直接浏览定义业务逻辑的源代码就不是什么异乎寻常的事情。他们将可以验证业务规则，然后把检查结果直接反馈给开发者。

现在，我们已经了解到DSL可以给开发者和领域用户带来不少好处，下面来认识一下目前已在业界广泛使用的几种领域专用语言。

1.3.2　流行的几种 DSL

DSL应用非常广泛。我敢肯定你开发过的每一个应用程序都用到不少DSL，虽然有些不一定被打上DSL的标签。表1-1列举了几种最常用的DSL。

表1-1 常用的DSL

DSL	用 途
SQL	关系型数据库语言，用于查询和变更数据
Ant、Rake、Make	几种用于软件系统构建的语言
CSS	样式表描述语言
YACC、Bison、ANTLR	几种用来生成语法分析器的语言
RSpec、Cucumber	Ruby环境下的行为驱动测试语言
HTML	用于Web的标记语言

你平时经常用到的DSL肯定不止这些。你能分辨出它们有什么共同特征吗？来看下面几点。

- 所有DSL都有对应专门的领域。每种语言都拥有“有限表达力”（limited expressivity），而你只能用一种DSL解决其特定领域的问题。你不可能只用HTML搭建出一套货运管理系统。

定义 马丁·福勒（Martin Fowler）用“有限表达力”来描述DSL最重要的特征。在2009年DSL开发者会议的主题演讲（参见1.9节文献[3]）上，他提出有限表达力是DSL与通用编程语言的根本区别。通用编程语言可以给任何事物建模，而一种DSL只能给一个专门领域建模，可是表现力更强。

- 使用表1-1列出的语言（以及其他被广泛使用的语言），一般即使用它们所建立的抽象。在绝大多数情况下，你并不需要了解语言的底层实现。每种DSL都提供了一套供你搭建解答域模型的契约，为了搭建更复杂的模型可以把不同的契约组合起来，但终归不需要跨出契约的范围和深入DSL的实现层次。
- 任何一种DSL都具备充分的表达能力，足以使不懂编程的用户理解程序的意图。DSL并非给开发者提供的一套API而已，它的每一个API都以领域语汇精炼地表达丰富的含义。
- 用任何一种DSL编写的源代码文件，即使数月之后再重新翻看，你也可以立即领会当初的意思。

事实证明，依托DSL进行开发更能鼓励开发者与领域专家进行更好的交流。这是其很重要的优点。借助DSL，不擅长编程的领域专家不必勉强转变为一般程序员。得益于DSL的表现力和其特意以沟通为目的提供的API，领域专家可以理解解答域的抽象实现了哪些业务规则，以及其实现是否充分覆盖了所有可能的业务场景。

我们来看一段有启发意义的Rakefile代码片段，示例中所用的Rake也是表1-1所列的一种DSL，主要用于构建Ruby语言编写的系统：

```
desc "Default Task"
task :default => [ :test ]

Rake::TestTask.new { |t|
  t.libs << "test"
```

```
    t.pattern = 'test/*_test.rb'
    t.verbose = true
    t.warning = false
  }
```

这段代码建立一系列单元测试，并把它们作为默认任务来运行。即使你不懂Ruby，也不会看错这段代码的意思；它的表达能力没有受影响。这是怎么做到的？它各处重要部分的用词正好匹配了你所熟悉的语汇，而且给DSL使用者提供了简单明了的界面。Rake的使用者是软件开发人员，所以这种语言将其语义设定到符合开发者预期和理解力的抽象层次。同样，如果打算开发一种给金融交易员群体使用的DSL，你必须谨记使表现力层次符合交易台上的人的预期和经验。这里的附加内容简要说明交易系统的一些基本术语；请了解一下相关定义，因为这些概念会反复出现在本书所用的DSL示例中。


金融中介系统：交易和结算

交易在两方（交易双方）之间进行，遵照交易市场的规章进行证券与现金之间的互换。交易只是承诺，需要在交易发生后的规定天数内完成结算。进行结算的日期称为结算日，根据若干因素确定，如实行交易的具体市场、证券的生命周期、交易的性质、实行交易的日期（交易日）等。

每一笔交易都对应一个现金价值。现金价值是购买证券的一方应付出的金钱数量。现金价值取决于若干因素，例如基本价值、印花税、经纪费用和佣金等。

交易在证券交易所成交后，其详情被输入到交易机构的后台完成一个交易充实过程，由交易系统计算出所有的细节事项：结算日、交易税、佣金和最终的现金价值。

设计DSL的时候，你要时刻把使用者放在心上。DSL的表现力和粒度要尽力满足用户理解的需要。你会在后面的章节中学习如何在用户感觉最自然的抽象层次上设计DSL。现在我们先来考虑DSL怎样更好地在问题域和解答域之间建立映射，填补图1-2缺失的一些环节，使你对此有更全面的认识。

1.3.3　DSL 的结构

图1-3展现了DSL脚本怎样将共通语汇联系到解答域的实现模型。

设计得当的DSL应该体现以下三项原则，以便与领域用户更好地“沟通”。

- DSL要为问题域制品提供直接的映射。如果问题域有一个名为Trade的实体，那么DSL脚本就必须包含同样名称同样角色的一个抽象。
- DSL脚本必须使用问题域的共通语汇。这些语汇将成为开发者与业务用户增进交流的催化剂。如图1-3所示，当业务用户与软件中的领域模型交互的时候，DSL脚本就是他们的用户界面。
- DSL脚本必须对底层实现进行抽象。这是抽象设计的一项重要原则，对于DSL的设计同样

适用。DSL脚本中不可以出现因为实现细节而引入的非本质复杂性[①]。

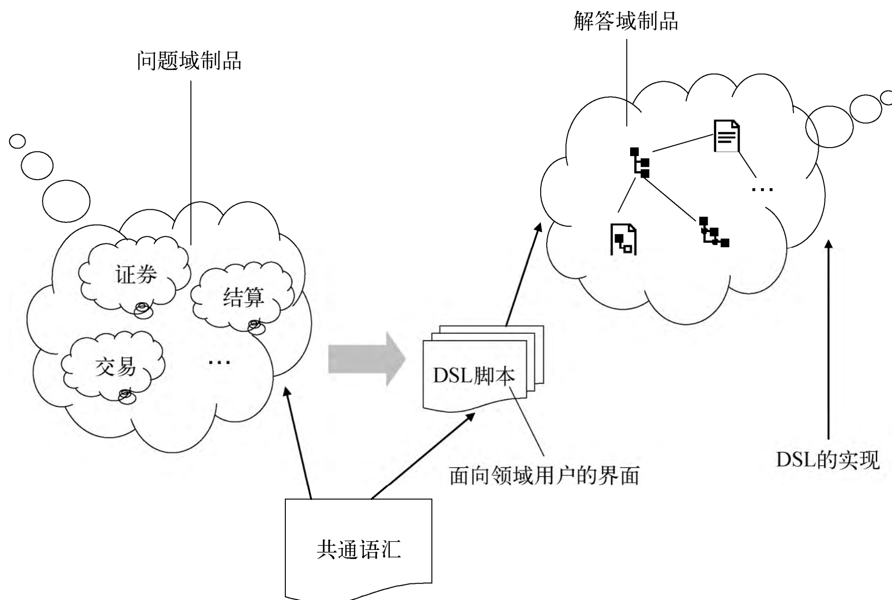


图1-3 DSL脚本将实现模型表示为领域语言。脚本中的用词都出自共通语汇，使用户对语言感觉更自然

在图1-3中，“DSL脚本”节点与其他节点的联系即为以上三项原则的形象表示。只要在设计中牢记这些原则，你所设计的DSL就能充分发挥与领域用户“沟通”的效果。下一节将讲述DSL的执行模型——当用户运行软件时DSL脚本及其实现模型是如何呈现给用户的。

1.4 DSL 的执行模型

领域专家通过DSL脚本理解领域模型和业务规则，而开发者负责实现DSL这个技术支撑平台。大多数情况下，DSL无非是覆盖于宿主语言之上的一个抽象层，向业务用户提供领域友好的界面。（其实不一定是宿主语言，详见1.5节的DSL分类。）可以这么说，你要做的事情就是对宿主语言进行扩展，在其上实现另一种语言。这种用一种语言实现另一种语言的概念有时候称为*元语言学抽象*。有些DSL并不通过内嵌方式实现，也许开发团队会为DSL特别设计一种定制语言。不同类型的DSL实现方式我们放到1.5节讨论，现在先来看看怎样执行DSL脚本。

图1-4展示了三种最常见的DSL脚本执行方式。

① 非本质复杂性（accidental complexity），与本质复杂性（essential complexity）相对应，指程序开发过程中出现的、与问题本身无关的复杂性。本质复杂性是与生俱来不可避免的，而非本质复杂性却是由于解决问题的手段而产生的。——译者注

(1) 脚本中的解答域模型可以直接执行，无需进行代码生成或其他变换操作。可能有一个解释器直接解释并运行脚本。UNIX中的微型编程语言awk和sed都是能直接执行的DSL。

(2) 在虚拟机上开发的DSL脚本遵照第二种执行模型。任何Java DSL脚本的语义模型都生成在JVM上执行的字节码。

(3) 有些语言提供编译时元编程能力。当用这样的语言开发DSL时，开发者在源代码中加入一些元结构，它们会在运行前被转译成一般语法成分。Lisp通过宏来支持这种实现手法，Lisp宏会在宏展开阶段展开成一般的Lisp成分（详见附录B）。对于这类语言，在为虚拟机生成字节码之前，存在一个对源代码进行转译的中间阶段。

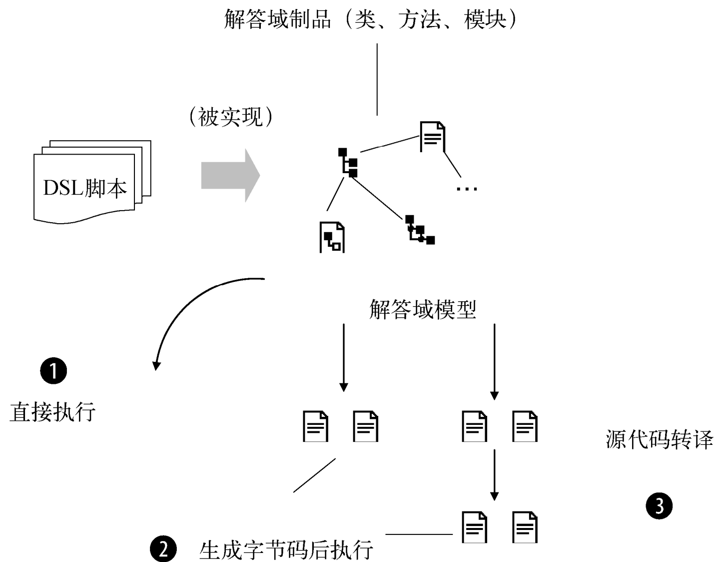


图1-4　DSL脚本的三种执行模型。实现了解答域模型的程序可以直接执行❶；可以编成字节码后执行❷；可以先对源代码进行转译（像Lisp宏），然后生成字节码再执行❸

熟悉三种最常见的DSL脚本执行模型之后，我们回头看看代码清单1-1中老鲍摆弄的那段DSL。你会发现，不管选择什么样的实现语言，DSL脚本下面必定有一个语义模型作支撑。这个语义模型可以是Ruby或Scala之类的宿主语言，也可以是工作在蹦蹦高证券公司的程序员专为金融交易DSL设计的一种定制语言。

以常用的构建工具Ant为例，它向用户提供了一种基于XML的DSL。当看到下面的XML片段时，开发者会认为它表达的都是一些熟悉的概念。我们很容易看懂它的任务目标，即构建一个`jar`，而这个任务依赖于`compile`任务。

```
<target name="jar" depends="compile">
  <mkdir dir="${build.dist}"/>
  <jar jarfile="${build.dist}/${name}-${version}.jar">
    <fileset dir="${build.classes}" includes="**"/>
```

```
    <fileset dir="${src.dir}">
      <include name="*"/>
    </fileset>
  </jar>
</target>
```

这段DSL脚本的背后有一个语义模型；其实现用Java类、方法和包的形式建立“任务”、“依赖”等界面。开发者在使用Ant时不需要越过DSL接口去深究Ant的内部实现；当然，因为Ant是一个可扩展的框架，所以还是存在例外情况，但也仅仅不过是例外。

目前为止，我们讨论的DSL脚本主要是通过扩展宿主语言来设计的，但DSL脚本并不只这一种类型。DSL还可以按其实现方式进行分类。下一节将阐述DSL的分类法。

1.5 DSL 的分类

DSL用领域语言来表达。领域的内涵越丰富，DSL的表现力就应当越强。对于领域用户来说，DSL帮助他理解领域的来龙去脉，至于开发者怎么实现其底层模型，这一点并不重要，只要DSL脚本能提供他对领域抽象的一致访问就行了。

最常见的分类方法是按照DSL的实现途径来分类。马丁·福勒曾将DSL分为*内部*和*外部*两大类，他的分类法得到了绝大多数业界人士的认可和沿袭。内部与外部之分取决于DSL是否将一种现存语言作为宿主语言，在其上构建自身的实现。内部DSL也称*内嵌式*DSL，因为它们的实现嵌入到宿主语言中，与之合为一体。（内部DSL在第5章和第6章有进一步的叙述，届时我们将演示如何用Ruby、Groovy、Scala、Clojure等JVM语言实现DSL。）外部DSL也称*独立*DSL，因为它们是从零开始建立起来的独立语言，而不基于任何现有宿主语言的设施建立。第7章和第8章将继续介绍外部DSL。

除了这两大类，还有新出现的一些DSL开发范式。例如，Intentional Software（http://www.intentsoft.com/）等公司推出了创建非文本型DSL的工具。像这样的一些发展和增长趋势详见第9章。目前我们暂且把注意力放到两个主要类别上，通过一些例子来看下它们各自的特点。

1.5.1 内部 DSL

*内部*DSL将一种现有编程语言作为宿主语言，基于其设施建立专门面向特定领域的各种语义。目前使用最广泛的一个内部DSL是Rails，它是在编程语言Ruby的基础上实现的。当你编写Rails代码，实际上就是在运用Rails给Web开发制定的各种语义编写Ruby程序。大多数情况下，内部DSL就是在宿主语言之上实现的库。2.1节将以Java和Groovy为宿主语言带你开发一种用于订单处理的DSL。图1-5描绘了内部DSL的结构。

如图1-5所示，内部DSL脚本只是在用宿主语言实现的抽象上进行了少许修饰。我们再来看看外部DSL。

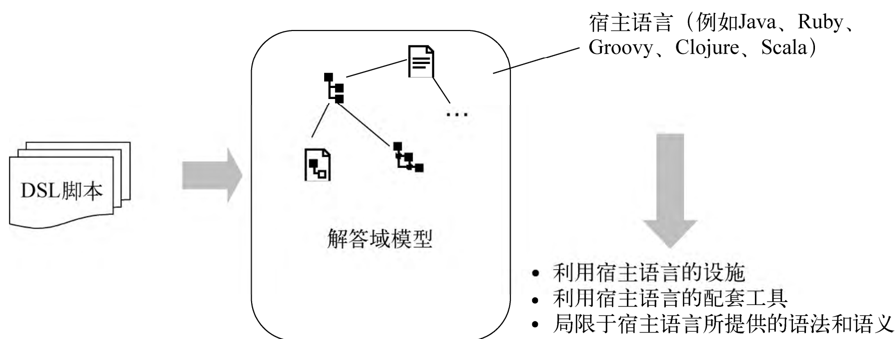


图1-5　利用现有宿主语言及其设施来实现内部DSL

1.5.2　外部 DSL

外部DSL是从零开发的DSL，在词法分析、解析技术、解释、编译、代码生成等方面拥有独立的设施。开发外部DSL近似于从零开始实现一种拥有独特语法和语义的全新语言。构建工具make、语法分析器生成工具YACC、词法分析工具LEX等都是常见的外部DSL。当然，外部DSL实现的复杂程度取决于你希望它有多丰富的内涵。一般来说，外部DSL并不需要像完善的语言那么复杂。第7章和第8章会给出大量示例。图1-6展示了外部DSL的结构是如何基于定制的语言设施形成的。

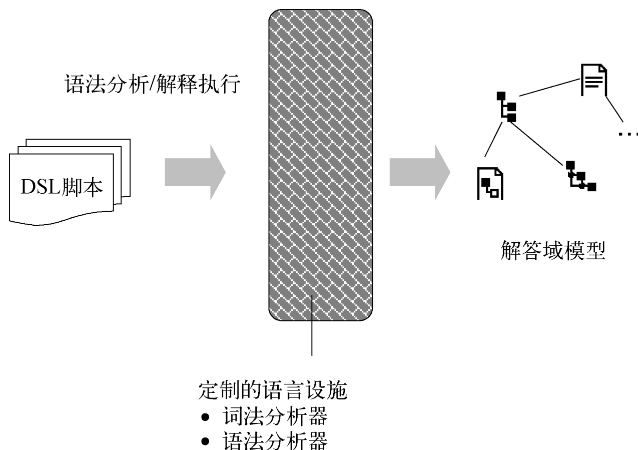


图1-6　你需要为外部DSL自行开发语言处理的设施。设施包括一般高级语言实现中常见的词法分析器、语法分析器和代码生成器等。注意，各部分的复杂程度取决于语言的精细程度

图1-6展示了外部DSL的一般组成部分。在实际开发中，上面列出的各部分未必全都用得上，而且你可能需要根据语言的复杂程度决定合并其中一些组成部分。

DSL是不是总要以文本的形式出现呢？不一定，很多时候图形化的表示更加一目了然。详情请看下文。

1.5.3 非文本 DSL

除了内部和外部DSL，业界还有一种正在增长的趋势，即倾向于发展更丰富的领域建模手段。DSL是领域的一种表示形式，但其定义中并没有硬性规定这种表现形式或语言必须是文本形式的。实际上，很多人都认为程序代码用作表达领域知识的媒介太过单薄，有时无力胜任。他们常给出以下理由：

- 文本允许的标识符号有限，限制了对领域问题的表达自由；
- 很多领域问题可以通过电子表格、图形化模型等丰富的制品形式更好地展现给领域用户；
- 在基于文本的脚本中，领域逻辑常散落在曲折交错的语法结构里，不经意地增加了复杂性；
- 领域专家操作起形象化的模型总是比操作源代码更自如。

出于以上原因，一种新型的DSL正迅速成为收集和建模领域知识的新一代手段。领域用户通过一种“投影编辑器”（Projection Editor）观察和处理领域知识的表现形式。投影编辑器可以将领域的适当视图投影给用户，用户可对其做各种调整，无需编写哪怕一行代码。然后，投影编辑器在后台生成代码对用户的意图建立模型。Intentional公司的DSL Workbench（http://www.intentsoft.com）和JetBrains公司的Meta Programming System（MPS，http://www.jetbrains.com/mps）是两种可以建立丰富形态DSL的建模工具。第9章讨论基于DSL进行开发的未来趋势，列举更多这方面的例子，也更详细地介绍了这类工具提供的功能。

把DSL分成内部、外部、非文本的DSL三大类，只是广义地看待不同的DSL实现方式。若从实用性出发，你可以把非文本DSL完全看做外部DSL，因为用来实现其API的底层设施并非宿主语言。

现在我们对什么是DSL有了相当的认识，也知道如何用它们增进开发者与领域用户的沟通，那么在什么情况下需要创造一种DSL？应不应该每开发一个程序都编写一种DSL？还是说存在一些特定的条件，在那样的情况下采用基于DSL的开发特别有利？

1.6 何时需要 DSL

每个应用程序的业务规则都应该明确、可读、直白。DSL是对业务规则建模的最佳手段。开发一种DSL，把原来写成`time() - 1209600`的时间表示形式变为`2.weeks.ago`并不难，但它对用户可能产生巨大的影响。

你应不应该在下一个项目中使用基于DSL的开发？作决定之前，你应该先掂量一下各种优缺点。DSL跟任何一种技术一样，有其暗藏的危险。身为开发者，你比任何人都更有资格判断眼前的问题是否需要用DSL来建模。为此，你需要了解DSL通常会有的一些优点和缺点。

1.6.1 优点

基于DSL的开发在领域复杂度较高时可以提供更高的回报。前面提过，几乎你经手的每个项目都会用上一些小的DSL引擎。当你开始规划一个复杂的建模项目时，应该在有意识地权衡过各种选择之后再做最后决定。下面提供的一些论点将有助于你决定是否采用基于DSL的开发方式。

1. DSL更具表现力

DSL趋向于提供一种覆盖面小、范围集中的API接口，处理的各种抽象概念都具有领域内的精确语义。用户热爱DSL。

2. DSL更精炼

因为精炼，DSL易于观看、观察、设想、展示。Dan Roam（参见1.9节文献[2]）称之为视觉化思考的四个步骤 。DSL的精炼性缩短了程序与问题之间的语义距离。

3. DSL设计于更高的抽象层次

DSL不需要应对低层次的语言构造、数据结构优化及其他实现手法。相反，DSL在一个更有利的层次体现领域知识，比起一般基于通用编程语言的实现层次，更便于领域知识的保存、验证和重用。这个特点使得DSL符合许多不懂编程的领域专家的需求。

4. DSL的回报更高

从开发生命周期的长远来看，基于DSL的开发趋向于产生较高的回报。

5. 基于DSL的开发容易扩大规模

如果项目团队对于特定编程语言的掌握程度不一，可以让熟练的程序员先集中精力实现DSL，然后给其他成员使用。因为DSL的抽象层次较高，所以更易于学习掌握，可以充当一种扩充开发团队的载体。

任何一种技术范式都有其优点，基于DSL的开发范式也不例外。我们会在第3章继续讨论基于DSL的开发。下面是DSL通常存在的一些潜在危险，它们在开发项目中的出现会令你大为头疼。

1.6.2 缺点

DSL的所有缺点都可以关联到软件开发生命周期中招致额外成本的实现开销。

1. 语言设计很难

实现DSL是一种语言设计工作，而语言设计是复杂的任务，且无法凭人多取胜。顾及从零开始设计一种语言的词法和文法的复杂程度，大多数人选择将DSL寄身于别的高级语言之中。即便如此，设计工作仍然相当难，绝不是生手程序员可以胜任的。后面几章会谈到各种语言特性以及用它们实现内嵌式DSL的情况。

2. DSL需要前期投入

在项目中引入基于DSL的开发也会引入前期成本。只有当模型的复杂度适中，这样的代价才值得接受。在开发周期的后期阶段，当成本被摊平之后，这样做的好处会最终显现出来。

3. 使用DSL可导致性能隐忧

DSL有时候会给程序带来性能隐忧。毕竟，它又增加了一个间接层。作为项目经理，你要考虑部署规模、重用范围等因素，才能决定是否采用基于DSL的开发。

4. DSL有时缺乏足够的工具支持

任何开发方法都需要充分的工具支持才能在程序员团体中普及。工具支持包括很多方面，比如有无IDE集成、单元测试支持、语言工作台（language workbench）、性能分析支持等。如果你的DSL生成多种目标语言用于执行，那么各种语言之间的互操作性也是一个潜在问题。

5. “学不完的DSL”现象

任何一种外部DSL都要求开发者另外学习。内部DSL只要求开发者学习它在现有宿主语言之上营造的接口。开发者经常对又要学习一种新语言感觉厌烦，不仅因为没完没了，还因为新语言的用途很有限。

6. DSL可导致语言间的摩擦

通常开发一个应用程序要用到不止一种DSL。当结合使用多种语言时，人们往往担心最终的领域模型不能保持一致。DSL的组合使用并不简单，因为一般各个DSL都是彼此独立地发展起来的。如果不小心应对，语言的多样性可以导致各自为政的混乱状态。

从图1-3可以看出，DSL是建立在实现模型基础上的语言学抽象。你把领域模型抽象得越好，就越容易在上面设计出自然的语言。我们来看看模型应该具备什么特质，才能为创造一种富于表现力的DSL奠定坚实的基础。

1.7 DSL与抽象设计

本章前面，我用“抽象”这个词泛指来自领域、表现出一组联系紧密的行为的任何制品。*抽象只关注对象的本质属性，不把任何不必要的细节呈现给用户*。但哪些才是本质的部分？这取决于你从什么样的观察角度去看待抽象。本节将介绍抽象与DSL设计的相关性，以及它对于DSL的表现力有何影响。

你将在第5章和第6章了解到，设计得当的抽象是搭建DSL的语言学构造的基础。那么，怎样才能使抽象设计得当？

在评价抽象好坏的各种判断标准中，我认为有四种基本特质是抽象设计最应该具备的。表1-2总结了这四种特质。

表1-2 良好抽象应具备的特质

抽象的特质	对设计的影响
极简	只公开那些向客户承诺过的行为。公开得越多，越容易暴露抽象的内部实现，而这会为后面的工作招致困难
精炼	保证抽象的实现不包含任何非本质的细节
扩展性	保证抽象设计可以在不影响现有客户的前提下渐进式发展
组合性	你所设计的抽象要可以和其他抽象进行组合，构成更高阶的抽象

怎样设计出好的抽象？这是另一个话题了。我不打算在此讨论得太详细，以免偏离本章的主题。关于抽象设计的详尽内容参见附录A。在那里，我会用大量真实示例深入分析表1-2中列出的每一种特质。请读者在阅读下一章之前先看一遍附录A。当你能够轻松辨别抽象设计的好坏时，将更加了解好的抽象设计对于助力各种DSL设计技巧发挥效力的益处。

1.8 小结

本章对DSL背后基本原理的冗长介绍至此接近尾声。在对一个具体领域建模的时候，你的实现要用该领域的语汇来表达。有了共通语汇作为媒介，DSL可以把领域的语法和语义带进你的解答模型。

DSL应当有足够的表现力，这要靠发挥宿主语言的威力设计出恰当的抽象。抽象的设计是一个迭代过程，DSL的设计过程也一样。你不可能第一次迭代就得到一种设计完善的DSL，它必定是经过开发者和领域专家的协作与努力逐渐形成的。请让领域专家尽早参与开发过程。如果领域专家能理解抽象的含义，能验证业务规则的实现，那就证明你的模型是正确的，而且具有足够的表现力。

为陌生的开发范式打基础一向是个艰辛的过程。恭喜你成功完成了任务。接下来，你将深入接触现实世界中的DSL设计与实现。第2章将偏重介绍一些真实存在的DSL，它们是用JVM平台上的几种现代语言实现的。这一部分内容首先从Java开始讲起，然后是表达能力更强的Groovy、Scala和Ruby。你会发现，在代表了当前发展方向的几种编程语言的帮助下，模型的表现力会相应提高。敬请关注！

要点与最佳实践

- ❑ DSL是**开发者和业务人员之间的交流媒介**。DSL的设计工作必须有领域专家参与。
- ❑ DSL**不一定适合所有情况**。请在衡量过各种有利和不利因素之后，再决定是否投入设计与开发。
- ❑ DSL的**设计过程必定是迭代的**。请为它付出应有的努力。
- ❑ 谨记DSL的**语法必须满足最终用户对表现力的要求**。不要过度设计DSL，那只会使语法变得庞杂并增加实现的复杂性。

1.9 参考文献

[1] Coplien, James O. 1998. *Multiparadigm Design in C++*. Addison-Wesley Professional.

[2] Roam, Dan. 2009. *The Back of the Napkin: Expanded Edition*. Portfolio Hardcover.

[3] Fowler, Martin. Introducing Domain-Specific Languages. 2009 DSL Developer’s Conference（http://msdn.microsoft.com/en-us/data/dd727707.aspx）.

第 2 章 现实中的DSL

本章内容

- 初试设计——第一个基于Java的DSL
- 利用Groovy改善DSL的表现力
- DSL的实现模式
- 选择合适的DSL类型

在第1章里，你已经看到DSL对于开发团队与领域专家之间沟通的改善作用。我们还介绍了DSL的总体架构以及其支持的不同执行模型。然而，如果缺少有意义的真实用例，空谈DSL又有什么用呢？在实际的项目中，你凭什么判断设计DSL是比使用传统软件开发模型更优的解决方案？本章就向你揭示真实世界中的DSL设计。

首先，我们用一个启发性的DSL示例演示从初始设计、实现到最后优化的真实过程；示例照旧来自金融中介业务领域。我们先探讨几种实现，然后阐释在设计DSL实现的过程中会遇到的一般模式。图2-1展示了本章探索过程的路线图。

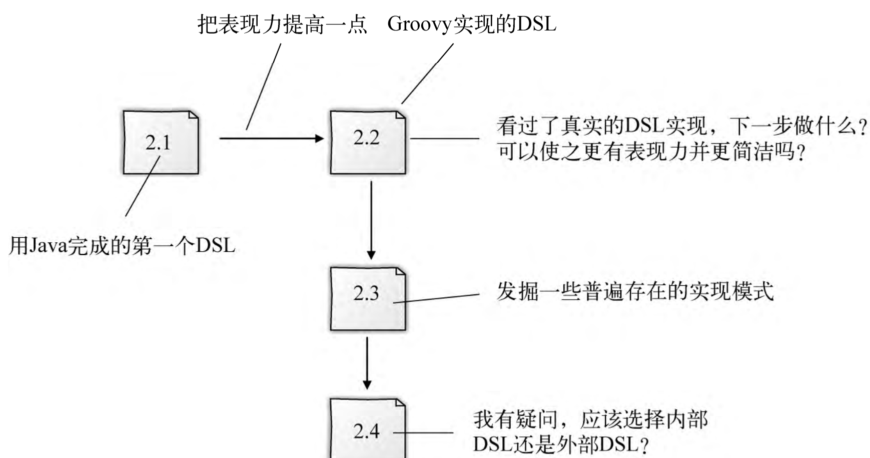


图2-1　第2章内容的路线图

本章每一节都会讨论真实世界中的DSL应用实例，它可能以实现用例的形式出现，也可能是你日后可以效仿的一套模式。通读本章，你将学会在对问题域建模时用基于DSL的编程范式思考问题。我们对照一个按照典型API思路设计的模型和一个按照DSL思路设计的模型，你会发现后者对于领域用户来说是更具表现力的表达形式。

2.1　打造首个 Java DSL

一例胜千言。第1章提到过，本书中的例子主要来自金融证券领域，作为对实现背景的交代，讲解中特地对领域概念给出注解。（请务必阅读补充内容中给出的领域知识。）这些概念注解除了帮助你理解特定领域，还是你学习DSL实现示例时的参考资料，方便你翻查例子中涉及的概念。因为同一个领域贯穿了前后的DSL示例，你可以不断完善和丰富它们。

在1.3节，我们看到交易员老鲍摆弄了一个DSL代码片段，对客户交易单做提交到交易所之前的各种处理。为了你即将开发的第一个DSL，我们再来丰富一下这个场景。

假设你负责实现一种用来处理交易单的DSL，其中的领域语汇与老鲍所用的差不多。看过第1章之后我们知道，领域专家是推进DSL开发的一种主要驱动力。如果有一种表现力充分的DSL，领域专家就能够理解开发团队实现的业务规则和逻辑，能够在代码离开开发实验室之前进行验证，甚至可能会以DSL用户的身份参与编写功能测试套件。专家们丰富的领域知识可以保证测试覆盖巨细靡遗，不仅如此，DSL经过领域专家的阅读和使用等同于经受了一次不折不扣的可用性测试。身为项目领导者，你务必要及早安排像老鲍那样的人物参与到开发中来。



金融中介系统：处理客户交易单

第1章提过，交易过程关系到在市场上买入和卖出证券，期间要遵守证券交易规则。交易始自投资者通过注册中介发出的交易单指令，这里的中介可以是股票经纪人、清算银行或者财务顾问。客户发出的交易单指令一般会包括若干信息，如打算交易（买入或卖出）哪种证券、多少数量、单位价格等。这些信息反映了交易双方对成交价格所作的限制。从下单到通知成交包括以下步骤：

(1) 投资者向中介下交易单指令；

(2) 中介记录交易单并转发给证交所；

(3) 交易单被执行，中介接到发回的成交通知；

(4) 中介记录具体的成交信息，并将通知转给投资者。

假设你的任务是实现一段DSL代码，用来针对具体的客户指令生成新的交易单。毋庸赘言，这种语言必定由领域语汇构成，并且在有效业务规则的语义约束下，允许用户（团队中的老鲍）任意组合交易单处理规则。不必一开始就纠结于最佳的语法设计，因为我们在第1章就说过，DSL必须迭代式演进，从来不可能一蹴而就。接下来你会看到我们的交易单处理DSL在逐步演进、通过选择不同的实现语言可使其获得更强的表现力，而我们将最终选择一种令老鲍满意的语言。虽

然说项目的第一步不应该把目标和期望定得太大太高，但我们从第1章了解到，创造一种DSL首先必须在所有项目干系人之间确立共通语汇。

2.1.1　确立共通语汇

老鲍观察了一下问题域，很快就圈定核心需求，随后三两下总结出交易单处理DSL必不可少的语言结构成分，如表2-1所示。

表2-1　初步总结交易单处理DSL的语汇

领域概念	明　细
(1)新交易单	■ 必须说明票据的名称 ■ 必须说明数量 ■ 必须说明是买入还是卖出 ■ 可以指定某交易单为“全部完成或放弃”（all-or-none），即要么接受交易单指定的全部交易，要么不交易。不存在部分成交
(2)交易单报价	■ 要求指定单位价格 ■ 可以用限价（limit-price）、限价收盘价（limit-on-close-price）、限价开盘价（limit-on-open-price）等形式设定单位价格
(3)交易单定价	■ 要求根据报价方案给整个交易单定价 ■ 报价方案可以从预设方案中选择，也可以由用户当场设定一个临时报价方案

有了这张语汇表，我们就可以着手实现了——选择Java这种占据统治地位的语言来完成第一次尝试。Java语言的开发者数量无出其右，不管用Java开发什么东西，这个庞大群体都是潜在的支持力量。除了动手实现，我们还要探讨Java作为实现语言在表现力方面的局限。老鲍自告奋勇帮我们编写功能测试和验证业务规则，我们的实现一定要让他觉得舒服才行。

2.1.2　用Java完成的首个实现

Java是一种面向对象（OO）语言，那么设计DSL的第一步自然就是把封装了客户交易单各方面属性的`Order`（交易单）抽象表达为一个对象。

1. 建立交易单抽象Order

代码清单2-1中就是老鲍即将用来处理新交易单的（用Java实现）`Order`类。

代码清单2-1　为Java DSL设计得的交易单抽象

```
public class Order {
  static class Builder {                       <- ❶ Builder设计模式
    private String security;
    private int quantity;
    private int limitPrice;
    private boolean allOrNone;
    private int value;
    private String boughtOrSold;
```

```
    public Builder() {}
    public Builder buy(int quantity, String security) {
      this.boughtOrSold = "Bought";
      this.quantity = quantity;
      this.security = security;
      return this;
    }
    public Builder sell(int quantity, String security) {
      this.boughtOrSold = "Sold";
      this.quantity = quantity;
      this.security = security;
      return this;
    }
    public Builder atLimitPrice(int p) {
      this.limitPrice = p;
      return this;
    }
    public Builder allOrNone() {
      this.allOrNone = true;
      return this;
    }
    public Builder valueAs(OrderValuer ov) {
      this.value = ov.valueAs(quantity, limitPrice);
      return this;
    }
    public Order build() {
      return new Order(this);
    }
  }

  private final String security;
  private final int quantity;
  private final int limitPrice;
  private final boolean allOrNone;
  private int value;
  private final String boughtOrSold;

  private Order(Builder b) {
    security = b.security;
    quantity = b.quantity;
    limitPrice = b.limitPrice;
    allOrNone = b.allOrNone;
    value = b.value;
    boughtOrSold = b. boughtOrSold;
  }

  //获取方法
}
```

❷ 用方法链接手法实现的连贯接口

❸ 不可变属性

这个类的实现代码用了一些常见的Java惯用法和设计模式来增强其API的表现力。Builder模式❶方便API的使用者分步完成交易单对象的构造。该模式还结合了连贯接口❷的设计手法，把领域问题用更易读的方式呈现出来。（连贯接口将在第4章详细讨论。）借助一个可变的builder对象，`Order`抽象的数据成员获得了不可变性❸，有利于实现并发。使核心抽象获得不可变性，正是通过builder来构造对象的一个正面作用。

定义 Builder设计模式常用于分步构造对象。它分离了对象的构造过程与对象的表示，所以不同的对象表示可以共用同样的构造过程。详情参见2.6节中的文献[5]。

Builder模式的实现部分就说到这里，以后我们再回头讨论代码中存在的一些问题。现在，我们先看看这个实现会给老鲍带来什么样的DSL。

2. 构造交易单

下面是一段交易单buider的应用实例，其中领域语汇的密度非常高；示例中由API形成的语言里，几乎可以找到表2-1列出的全部关键字：

```
Order o =
  new Order.Builder()
    .buy(100, "IBM")
    .atLimitPrice(300)
    .allOrNone()
    .valueAs(new StandardOrderValuer())    ❶ 交易单定价算法
    .build();
```

虽然我们的DSL已经用了正确的语汇，但本质上还是一段Java程序，仍然脱不开Java编程语言语法上的限制和琐碎的缺点。调用`valueAs`的时候❶，你需要指定一个定价算法作为它的输入，但算法只能在当前上下文以外实现。这是因为Java不直接支持高阶函数，所以我们没办法优雅地就地定义定价策略。这个Java实现的用户只能预先定义每一种交易单定价策略的具体实现。我们把“交易单定价”的契约实现为一个接口：

```
public interface OrderValuer {
  int valueAs(int qty, int unitPrice);
}
```

在Java中模拟高阶函数

虽然Java不直接支持高阶函数，但是有一些库用对象模拟出这种特性，如lambdaJ（http://code.google.com/p/lambdaj）、Google Collections（http://code.google.com/p/guava-libraries）和Functional Java（http://functionaljava.org）。如果Java是你的唯一选项，但又希望对高阶函数建模，这几个库不失为可行的途径。但这些库提供的选项存在缺点，就是较为烦琐，而且优雅程度肯定不如Groovy、Ruby、Scala等语言直接提供的语言特性。

DSL的用户针对特定定价策略分别定义该接口的具体实现：

```
public class StandardOrderValuer implements OrderValuer {
  public int valueAs(int qty, int unitPrice) {
    return unitPrice * qty;
  }
}
```

这时候老鲍发现自己没办法在现场随时定义定价策略，我们的DSL满足不了他一开始提出的需求。这可是个大挫折，毕竟我们自诩DSL能让不懂编程的领域专家写出有意义的功能测试。另外，老鲍还对交易单处理DSL提出了几点意见，如下。

- **语法烦琐**　语言中含有太多括号等令人眼花缭乱的东西，容易干扰不懂编程的领域专家。
- **语法中与领域无关的复杂性**　老鲍指的是用户必须显式使用Builder类。其实，去掉Builder类这重复杂性也能实现DSL，只要把Order类的获取方法都改成链式方法，同样能构造出连贯接口。只不过，Builder类对抽象设计还有正面的影响，它促使我们设计出一种不含可变属性的不可变抽象。那么，能不能从语言中去除这些不必要的语法成分？我们可以再次运用抽象手段把builder隐藏起来，让表面上的语法看起来更直观：

```
new Order.toBuy(100, "IBM")
         .atLimitPrice(300)
         .allOrNone()
         .valueAs(new StandardOrderValuer())
         .build();
```

这个取巧方案仅仅将复杂性从语法推到实现中，但毕竟使烦琐的部分留在DSL的实现层面，使用起来简洁多了。

3. 分析Java DSL

对于代码中显式出现的Builder模式，Java程序员百分百认可它的作用，也赞赏它使API连贯流畅这一点。如果DSL的用户都是Java程序员，那么我们设计的这种基于Java的DSL还算令人满意。但不可否认，Java的语法烦琐是一个缺点，我们可以选择一种表现力更强，而代码更简洁的实现语言来克服这个缺点。下面我们就来详细分析Java代码，看看是Java语言的哪些特性导致了老鲍不愿看到的那些语法复杂性。表2-2罗列了老鲍报告的各种不足应该归咎的Java语言特性。

表2-2　Java DSL的不足和应该归咎的Java语言局限

DSL的不足	应该归咎的Java语言特性
烦琐（不必要的括号和语法成分）	■ 属于基本的Java语法 ■ 函数必须带括号。在对象和类上调用方法必须用点号
与领域无关的复杂性	■ Java不是一种可以自我扩展的语言。很多常见的惯用法必须通过额外的间接层（设计模式）来表达 ■ 抽象设计中需要构造一些额外的类结构，其中一些会在对外公开的API里冒出头来，成为表面语法的一部分。前面示例中的Builder类就属于这种不必要的语法障碍 ■ Java不是一种解释语言。执行任何Java代码片段都必须先定义一个带public static voidmain方法的类。不管怎么说，在DSL用户的眼中，这就是掺杂进来的语法噪音
不能当场定义定价策略函数	■ Java不直接提供高阶函数这种语言特性

接下来，我们将逐一探索能满足老鲍愿望的各种改进手段，使DSL对领域专家更加友好。

2.2　创造更友好的 DSL

DSL的表现力高低只能由用户来评判。老鲍已经指出我们的Java方案有好几个方面需要改善，必须更贴近问题域的要求。为了给老鲍创造一种更友好的DSL，我们来尝试两种方案。其一是增

加一个XML层，把领域语言外部化，将其表现为更适合人类阅读的形式。第二种方案是彻底改用一种表现力更强的编程语言来实现DSL——Groovy。

2.2.1　用XML实现领域的外部化

业务上人们常将XML用作标记语言，那么何不用它来设计我们的领域语言？XML有充分的工具支持，被所有浏览器和IDE认可，而且有大量框架和库可用于解析、处理和查询。

确实，领域专家可以脱离编程环境编写XML结构，在这个意义上讲XML可外部化。但XML完全是声明式的，语法又特别烦琐，还很难表达控制结构。代码清单2-1中的交易单处理DSL如果用XML来表述，差不多就是下面这段代码的样子。我已经刻意省略了一部分因为僵硬的XML语法造成的臃肿结构。

```
<orders>
  <order>
    <buySell>buy</buySell>
    <quantity>100</quantity>
    <instrument>IBM</instrument>
    <limitPrice>300</limitPrice>
    <allOrNone>true</allOrNone>
    <valueAs>...</valueAs>
  </order>
  ...
</orders>
```

XML本来不是用来编程的，而是用于表达一种完全可移植的文档结构。DSL往往需要包含一些控制结构，而XML很难优雅地表达这些结构。很多J2EE（企业版Java平台）框架通过XML提供声明式配置参数。但如果进一步将XML的用途推广到编写业务逻辑和领域规则，你很快就会遇到之前Java实现中存在的表现力瓶颈。与其这样迂回，还不如就在我们朝夕相伴的编程语言中间寻找解决之道。记住，语言才是我们最得力的编程工具。

2.2.2　Groovy：更具表现力的实现语言

你现在可能已经意识到，我们只是在底层实现语言的能力范围之内设计DSL。DSL用户所用的语言根本就是开发者用来实现DSL的语言。老鲍针对我们的第一次尝试指出的问题其实都是Java编程语言的固有局限，不可能在DSL实现中规避。说到底我们的实现技术就是在宿主语言中内嵌DSL，这一点已经在1.7节讨论DSL的内部和外部分类时说过。

所以，我们应该考虑一种比Java表现力更强的语言作为宿主语言。Groovy编程语言运行在JVM平台上，表现力强于Java，是动态类型语言，还支持高阶函数。

1. Groovy方案

不断阅读本书，你会看到Groovy有助于设计更优秀DSL的各种语言特性。下面来用Groovy实现交易单处理DSL。首先，我们来看一段用Groovy实现的DSL代码片段，它与之前的Java示例功能完全相同：

```
newOrder.to.buy(100.shares.of('IBM')) {
  limitPrice   300
  allOrNone    true
  valueAs      {qty, unitPrice -> qty * unitPrice - 500}
}
```

这段代码创建一个新的客户交易单，内容是购买100股IBM股票，限价300美元，购买方式为全部完成购买或全部放弃（all-or-none模式）。交易单定价按照代码中给出的公式计算。这段代码的执行结果和先前的Java示例是一样的；不过，Groovy实现高阶抽象的能力造成了表现效果的差异。因为Groovy具备超凡的元编程能力，我们才得以构造出像`100.shares.of('IBM')`这样的DSL语句结构，创造出让领域用户感觉更自然的语言。代码清单2-2中是用Groovy实现的交易单处理DSL的完整实现。

代码清单2-2　Groovy实现的交易单处理DSL

```
class Order {
  def security
  def quantity
  def limitPrice
  def allOrNone
  def value
  def bs

  def buy(su, closure) {
    bs = 'Bought'
    buy_sell(su, closure)
  }

  def sell(su, closure) {
    bs = 'Sold'
    buy_sell(su, closure)
  }

  private buy_sell(su, closure) {
    security = su[0]
    quantity = su[1]
    closure()
  }

  def getTo() {
    this
  }
}

def methodMissing(String name, args) {
  order.metaClass.getMetaProperty(name).setProperty(order, args)
}

def getNewOrder() {
  order = new Order()
}

def valueAs(closure) {
```

❶ 通过这个钩子拦截对不存在方法的调用

❷ 通过闭包实现定价策略的就地定义

```
  order.value = closure(order.quantity, order.limitPrice[0])
}

Integer.metaClass.getShares = { -> delegate }
Integer.metaClass.of = { instrument ->  [instrument, delegate] }
```

❸ 通过元编程手段注入新的方法

下面带你一起欣赏Groovy实现的妙处，不过暂时仅限于在这个特定实现中起到突出作用的几项语言特性。Groovy还有很多其他特性令它成为一种特别适合用来实现DSL的语言，我们等到第4章和第5章再作全面介绍。这个例子取得了如此出色的表现效果，我们看看这应该归功于哪些Groovy特性吧。

2. 通过`methodMissing`动态合成新方法

Groovy允许调用不存在的方法，而`methodMissing`作为钩子拦截所有这类调用❶。对于交易单处理DSL，每当调用`limitPrice`、`allOrNone`等方法，这类调用都会被`methodMissing`拦截下来，并被转换成调用`Order`对象属性的获取方法。`methodMissing`钩子既能节约代码又兼具灵活性，无需显式定义也可以添加新的方法调用。

3. Groovy元编程技术之动态方法注入

我们通过元编程技术向`Integer`内置类注入了若干方法，起到了增强语言表现力的效果。注入的`getShares`方法为`Integer`类增加了一个名为`shares`的属性，为构造自然流畅的DSL提供了非常有用的语言成分❸。

4. 直接支持高阶函数和闭包

这可能是令Groovy等语言在DSL表现力上压倒Java的最重要的语言特性。差别非常显著，只要比较一下Groovy和Java版本中的`valueAs`方法调用❷就能领会。

现在Groovy DSL的实现和应用代码都已就绪，但还差一个机制将它们集成在一起，而且需要建立一个能运行任意DSL代码的执行环境。我们来看看怎么做。

2.2.3 执行Groovy DSL

Groovy具备执行脚本的能力，它的解释器可以执行任意Groovy代码。你可以利用这一点为交易单处理DSL建立一个交互式的执行环境。我们需要把DSL的实现（代码清单2-2）保存成ClientOrder.groovy文件，把应用代码保存成另一个文本文件——order.dsl。注意，我们要保证两者的路径都在`classpath`中，然后向Groovy解释器输入下面的脚本：

```
def dslDef = new File('ClientOrder.groovy').text
def dsl = new File('order.dsl').text
def script = """
  ${dslDef}
  ${dsl}
"""
new GroovyShell().evaluate(script)
```

> **在核心应用程序中集成DSL**
>
> 本节中的示例只介绍了一种集成DSL实现和DSL应用程序代码的方式。我们将在第3章讨论于核心应用程序中集成DSL时介绍更多集成方法。

> 该示例使用字符串拼接方式来生成最终的执行脚本。这样的做法有个缺点，即如果执行中出现错误，栈跟踪信息中报告的行号会对不上源文件order.dsl中的行号。再次重申，DSL的建立和与应用程序的集成是一个迭代过程。第3章会探讨在应用程序中集成Groovy DSL的另一方法，然后会对此进行改进。

祝贺你！你已经成功设计并实现了一种令领域用户满意的DSL。基于Groovy的交易单处理DSL充分满足了对表现力的要求，远胜于之前的Java版本。更重要的是，你现在知道设计DSL是个迭代过程。假如当初没有开发Java版本，你会很难想象实现语言的表现力会有如此重要的影响。



本书第二部分（第4章~第8章）将介绍各种DSL实现语言，除了Groovy，还有其他JVM语言，如Scala、Clojure和JRuby。对比不同的语言实现，你会发现宿主语言的特性差异造就了多种多样的DSL实现技术。

从头到尾实现了一种DSL来解决真实案例，相信你已经全面了解了如何通过一连串的去芜存菁过程，步步为营地完善实现。Groovy实现最终被证明可以满足用户对表现力的要求，但良好的表现力可以归功于哪些底层实现技术呢？

为内部DSL选择适当的实现语言，你可以享受一些实现手段上的便利。只有灵活运用宿主语言的惯用法和从语言特性中衍生的便利技术，你才能把宿主语言塑造成优秀的DSL。我们在实现中利用了Groovy提供的一些技术，但并非所有DSL都相似。任何语言都有一定的设计模式，它们依赖于整体开发环境下的各种约束条件，如实现平台、团队成员的核心技能、应用程序的总体架构等。

下一节，我们就来看看DSL的一些实现模式。模式就像现成的设计知识，你可在自己的实现工作中重用，利用它们发掘宿主语言的力量去创造友好的DSL。你将会了解到，无论内部DSL还是外部DSL，它们都在具体的实现条件下表现出五花八门的模式。虽然你不可能在每一种语言中都实现所有这些模式，但必须全面地掌握它们，这样才能针对实现平台作出最有利的选择。

2.3　DSL 实现模式

DSL开发实践中存在数量众多的架构模式，简单地把DSL按内部或外部分类实在过于宽泛。内部DSL的共性是都建立在宿主语言之上。外部DSL的共性是都重新建立一套语言设施。起源上的共性并不足以全面刻画DSL分类体系。我们从第1章看到，DSL是一系列优秀抽象设计原则的化身。而设计优秀的抽象，不仅需要考虑各构成元素在形式上的共性，还要考虑每个元素展现出来的差异性。

接下来，我们将展示一些体现了差异性的实现模式。即使是同一类别的DSL也存在多种多样的架构模式，只有了解它们，你才能更好地针对DSL需求找到合适的具体实现架构。对那些一再出现的模式发掘得越多，你越容易重用自己的抽象。本书附录A讨论过这样的抽象设计原则，即当抽象可以被重用时，用它们设计出来的语言也会更容易扩展。如果你还没有读过附录A，请务必现在翻看。附录A中的知识对于本书后面的学习历程是很有帮助的。

2.3.1 内部DSL模式：共性与差异性

内部DSL其实随处可见。像Ruby和Groovy这类语言既有灵活而简洁的语法，又有强大的元编程模型，在语言的强力支持下，在用它们写成的软件中几乎都可以找到DSL的身影。所有内部DSL都有个共同模式，它们总是在现有宿主语言之上实现的。提到*内部*DSL的时候我更喜欢说成*内嵌*DSL，因为这可以突显它的一个架构特征。如果用不同的方式去运用宿主语言提供的设施，你创作出来的DSL也会在形式、结构、灵活性和表现力等方面表现出差异。

内部DSL主要有如下两种形态。

- **生成式**　领域专用的结构体经过编译时宏、预处理器或某种运行时MOP（Meta-Object Protocol，元对象协议）的转换生成实现语言的代码。
- **内嵌式**　领域专用的类型内嵌于宿主语言的类型系统。

即使这么细分也免不了有模棱两可的情况。比如Ruby及其Web框架Rails。Rails是一种用Ruby语言写成的内部DSL，可以说Rails*内嵌于*Ruby。但同时Rails又运用了Ruby的元编程能力，以便在运行时生成大量代码，所以它又是*生成式*的。

还有一类静态类型语言单纯将DSL内嵌于宿主语言的类型系统，Haskell和Scala是其中的代表。在这样的语言中，DSL继承了宿主类型系统的全部能力。另外，Haskell有一种语言扩展（Template Haskell）为语言增加了生成式能力，而这是通过宏来实现的。

仅在内部DSL这一类别下就有数量众多的不同形态模式，有的语言还同时支持多种DSL开发范式。图2-2展示了内部DSL的一个分类图谱，并且在每一种模式旁边标注了支持它的一些语言。

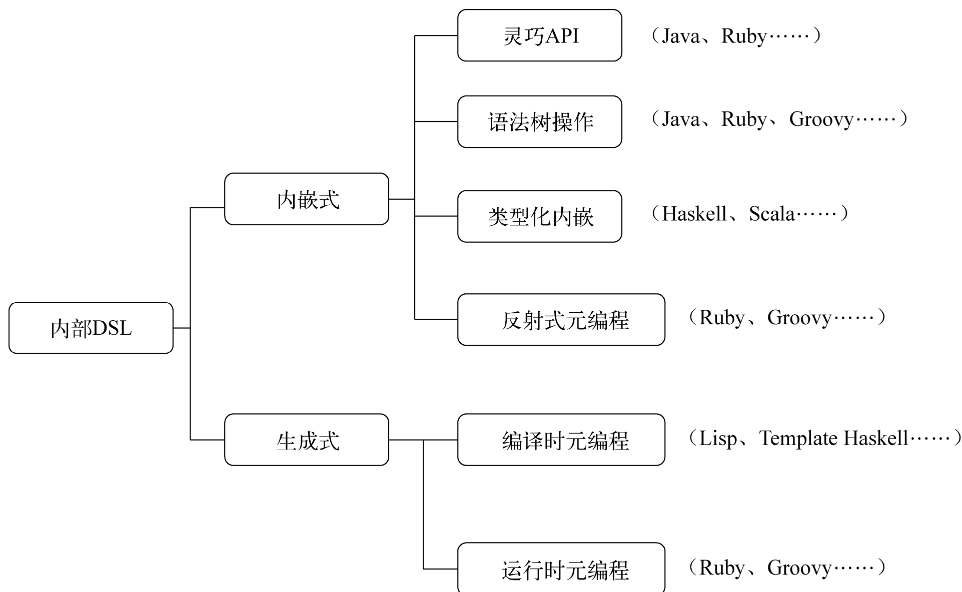


图2-2　内部DSL的各种实现模式，不太严谨的分类

下面我们就参考图2-2逐一介绍这些常见的内部DSL实现手段。



第4章和第5章可作为本节材料的补充阅读，这两章将更加详细深入地阐释DSL的模式与实现。

1. 灵巧API

灵巧API（Smart API）可能是最简单也最常见的DSL实现技术。这种技术的基础是方法串联，近似于实现Builder模式（参见2.6节文献[1]）。Martin Fowler将灵巧API称作*连贯接口*（http://www.martinfowler.com/bliki/FluentInterface.html）。在这种模式下，你所创建的API会按（要建模的）领域动作的自然序列串联起来。环环相扣的动作自然产生连贯接口，而具有领域含义的方法名可以方便领域用户的阅读和理解。下面的代码片段来自Guice API（http://code.google.com/p/google-guice/），它是谷歌开发的一个依赖注入（DI）框架。假设用户打算声明一个应用模块，把一个具体实现绑定到某Java接口，用Guice API写出来就是下面这样子，它看起来流畅自然，而且清楚地表达了用例的意图：

```
binder.bind(Service.class).to(ServiceImpl.class).in(Scopes.SINGLETON)
```

图2-3表现了API一环套一环的样子，调用得到的返回对象立刻又在其上发出另一个调用。

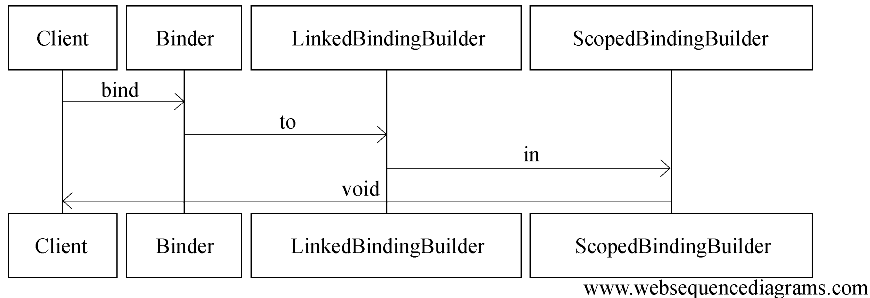


图2-3　以方法串联手法实现的灵巧API。方法调用一个接一个，并直到最后才返回给客户

通过方法串联手法，我们利用宿主语言的设施和领域语汇完成了灵巧API的构造。但这种手法有个缺点，它很容易导致大量可能没有太大独立存在意义的小方法。而且，不见得所有的用例都适合用连贯接口实现。若通过Builder模式增量构造并配置对象，一般方法串联的建模方式表现效果最好，2.1节中用Java实现的交易单处理DSL就是很好的例子。不过在Groovy、Ruby等语言中，由于其提供了“命名参数”特性，Builder模式和连贯接口显得有些多余。（Scala 2.8也支持带默认值的命名参数。）举个例子，刚才那段Java代码如果改成Groovy版本，只是混用一般的参数和命名参数而已，代码立即就简洁了许多，但表现力一点都不输原来的版本：

```
binder.bind Service, to: ServiceImpl, in: Scopes.SINGLETON
```

灵巧API是内部DSL常用的一种模式。具体的实现取决于所用的语言。这里有个需要牢记的要点：实现DSL模式的时候，应该总是选择最合乎语言习惯的实现手法。第4章讨论到连贯接口

在内部DSL实现中的作用时，我还会针对这个主题补充更多的例子和不同的实现方式。现在，我们接着看下一个模式。

2. 语法树操控

语法树操控也是用来实现内部DSL的一种手段。它按照Interpreter模式（参见2.6节文献1），利用宿主语言的设施来生成并操控AST（Abstract Syntax Tree，抽象语法树）。AST产生之后，开发者在语法树上遍历，安插、修改AST的结构，把根据领域逻辑希望得到的代码反映在AST的结构上。Groovy和Ruby都提供了这方面的设施，可以通过库来操作AST以生成代码。

你有没有想起来，这种实现手段其实是Lisp语言天生就具备的能力。在Lisp语言里，每个程序都是一个列表结构，而这些列表结构就是程序员可以操控的AST。Lisp宏所做的事情归根结底就是通过操控AST来生成代码。程序员可以用这种手段扩展Lisp语言的核心语法。

3. 类型化内嵌

以元编程为基础的DSL实现模式依靠代码生成技术保持语言界面的抽象层次，使之精确地满足领域要求。但如果选用的宿主语言不支持任何形式的元编程，该怎么办？在设计DSL的时候，你必须秉持极简的方针以决定哪些东西可以作为专用语法展现给用户，这是极重要的设计要求。宿主语言的设施提供的抽象手段越多，开发者越容易实现极简的目标。

静态类型语言把*类型*作为对领域语义的一种抽象手段，同时使DSL的表面语法简洁精炼。这种方式不通过生成代码来表达领域行为，而是以宿主语言的类型和操作为载体定义并实现领域专用类型。这些专用类型即构成用户面对的DSL语言界面；注意，用户并不在意类型背后的具体实现。类型化的模型在一定程度上隐式地保证了编程模型的一致性。图2-4简单说明了类型带给DSL的好处。

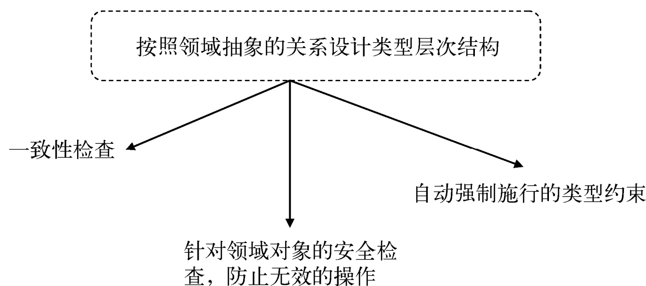


图2-4　嵌入了类型约束的DSL在很多方面隐式地保证一致性。请用类型对DSL抽象建模。在类型中定义的约束条件自动获得编译器的检查，检查甚至发生在程序运行之前

这种模式的最大优点在于：因为DSL的类型系统内嵌于宿主语言的类型系统，所以其类型系统会自动获得宿主语言编译器的语法检查。DSL的用户也可充分利用IDE集成的各种宿主语言功能，诸如智能提醒、代码补全、重构等。

下面的Scala示例是对`Trade`抽象的建模。该示例中，`Trade`、`Account`和`Instrument`都是封装了业务规则❷的*领域专用类型*❶。在Ruby或Groovy里面，领域行为借由生成额外代码实现；

在Scala里面，类似的语义实现于类型载体之上，并由编译器保障一致性。

```
trait Trade {
  type Instrument
  type Account

  def account: Account
  def instrument: Instrument

  def valueOf(a: Account, i: Instrument): BigDecimal
  def principalOf: BigDecimal
  def valueDate: Date
  //..
}
```

❶ 对领域类型的抽象

❷ 对领域操作的抽象

Haskell和Scala等语言具有高级静态类型化能力，使你完全可以设计出单纯的类型化内嵌式DSL，无需求诸于代码生成、预处理器或者宏技术。DSL用户可以通过语言本身实现的组合子，把类型化的抽象组织成DSL语句。这类语言的类型系统拥有一些高级功能，比如支持类型推导、高阶抽象等，可使语言简洁又不失表现力。Paul Hudak在1998年即用Haskell语言演示过这种DSL实现方式（参见2.6节文献[2]），当时他用Monad化的语言解释器设计、部分求值技术和多阶段编程（staged programming）技术来实现可以增量式演进的纯内嵌式DSL。Christian Hofer等人也在2.6节文献[3]中讨论了Scala的类似实现，甚至还论及如何巧妙利用traits、虚类型、高阶泛型、族多态等Scala特性在单一DSL界面下“多态地”嵌入多个实现。在第6章，我将用一些实现范例说明Scala的静态类型对于设计纯粹的EDSL（Embedded Domain-Specific Language，内嵌式领域专用语言）的帮助。

定义　Monad代表一种经由Haskell语言得到推广的计算模型。Monad让你按照预定义的规则去构建抽象。第6章在讲述用Scala实现DSL的内容时会探讨Monad化的程序结构。更详细的定义请参阅http://en.wikipedia.org/wiki/Monad_(functional_programming)。

接下来要讲述的几种常见DSL实现模式都属于元编程模式，其支持语言的兴盛与它们息息相关。在不同的DSL实现技术中，若论定制DSL语法的能力，元编程可谓当仁不让。

4. 反射式元编程

有些模式针对小范围的局部实现，比如灵巧API。而针对大范围的一般实现策略，也有一些模式可以遵循。后一类模式不但对实现的整体结构有决定性影响，而且本身就是宿主语言的关键特性。回顾我们讨论内部DSL实现模式的过程（参见图2-1所示的路线图），元编程的概念一再以各种面目现身于DSL设计之中。其中一种形态——反射式元编程——正是我们这一节所要讨论的模式。

假设你要设计一种DSL用来读取配置文件，然后根据配置内容动态地调用若干方法。下面是真实的Ruby示例，它从一个YAML文件读取方法的名称和参数，然后用读到的参数动态地调用方法：

```
YAML.load_file(x_path).each do |k, v|
  foo.send("#{k}", v) unless foo.send(k)
end
```

因为直到运行时才能得知方法名称，我们要用到Ruby的元编程能力，通过`Object#send()`语法动态地向对象发出消息，这有别于一般静态方法调用的点符号语法。这里所用的编码技巧就叫反射式元编程；Ruby在运行时获知方法，然后调用。在DSL实现中处理动态对象的时候，你就可以利用这种技巧推迟方法调用，直到从配置文件等途径收集到全部所需信息。

5. 运行时元编程

反射式元编程只能发现运行之前已经存在的方法，不过有的元编程形式能够在运行期间动态地生成新代码。运行时元编程也是精简DSL表面语法的一个途径。它使DSL外观上显得轻巧，把费力的代码生成工作被转移到宿主语言的后端设施去处理。

有些语言会将它们的运行时基础组件暴露出来，使之成为程序员可以操控的元对象。在Ruby或Groovy语言中，程序可以利用这些组件在运行时动态改变元对象的行为，或者注入新行为去实现领域结构。图2-5简单说明了Ruby和Groovy中元编程的运行时行为。

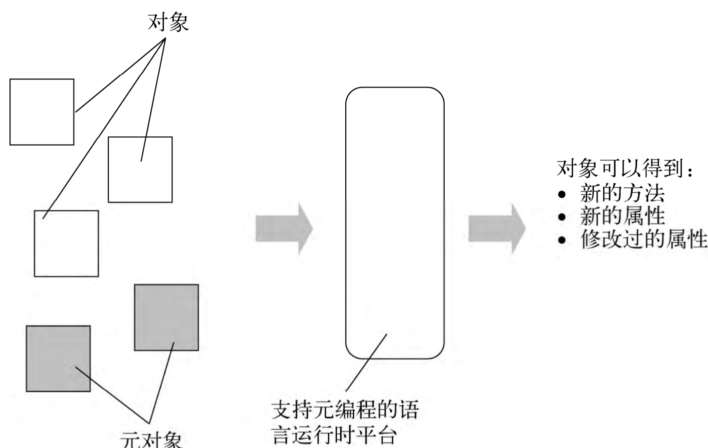


图2-5　支持运行时元编程的语言允许代码一边生成一边执行。生成的代码可以修改已有的类和对象，动态地增加新行为

我们在2.1节用Groovy开发过交易单处理DSL，当时就用到了这里所说的运行时元编程技术在内置类`Integer`上增加新方法`shares`和`of`。实际上对于DSL打算执行的动作语义，这两个方法并没有任何实质意义。它们只起到一种连结作用，目的是让语言更贴近建模领域。图2-6是一段用Groovy实现的DSL语句，图上为这一串连续调用的方法标注了每个方法的返回类型。新生成的方法贯穿于语句中间，大大改善了语言的表达能力，这些都是运行时元编程的效果。

Rails和Grails是两个特别强大的Web开发框架，它们都借助了运行时元编程的力量。在Rails中，你只要写下下面的代码，Ruby的元编程引擎会根据`Employees`表的定义生成所有相关的关系模型及验证逻辑代码。

```
class Employee < ActiveRecord::Base {
  has_many :dependants
  belongs_to :organization
  validates_presence_of :last_name, :title, :date_of_birth

  # ..
}
```

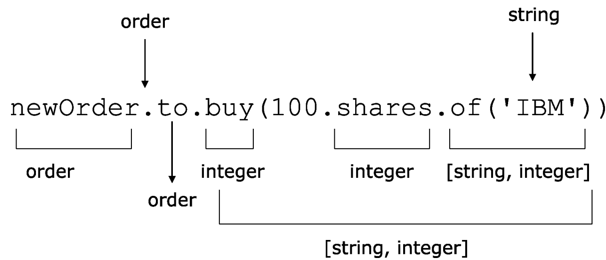


图2-6　通过运行时元编程获得丰富的领域语法

运行时元编程通过在运行时阶段生成代码使DSL动态化。不过，还有另一种形态的代码生成技术，它发生在编译阶段，因而不会给运行时增加任何额外开销。下面我们要讨论的DSL模式就是编译时元编程，这种模式大多出现于Lisp语言家族。

6. 编译时元编程

若采用编译时元编程，我们可以为DSL加上定制语法，这点跟刚刚讨论过的运行时元编程很像。虽然两种模式有相似的地方，但也存在着关键的区别，请看表2-3。

表2-3　编译时元编程与运行时元编程的对比

编译时元编程	运行时元编程
开发者定义的语法在运行时之前、编译阶段得到处理	开发者定义的语法在运行时期间经过元对象协议（MOP）处理
没有运行时的额外开销，因为到达运行时平台的都是正常形式的程序	有一定程度的运行时开销，因为要处理元对象和生成代码

在编译时元编程的典型实现中，用户通过与编译器的交互在编译阶段生成程序片段。



宏是最常见的一种编译时元编程实现途径。4.5节将通过Clojure示例深入解释编译时元编程的工作原理。

C语言中基于预处理器的宏，还有C++语言中的模板，都是能在编译阶段生成代码的语言基础结构。不过，如果追溯编程语言的历史，Lisp才是编译时元编程的鼻祖。C语言宏是在词法层面上进行文本替换操作。而Lisp宏直接对AST操作，在句法层面上提供了极强的抽象设计能力。从图2-7中的示意图可知，开发者定制的DSL专用语法经过宏展开阶段的处理变成普通的代码成分，然后被转发给编译器。

本章对内部DSL实现模式的讨论到此为止。我们介绍的几种元编程方式主要见于Ruby、Groovy、Clojure等动态语言。另外，我们介绍了静态类型化技术，以及它对于设计类型安全的

DSL的作用。第4章~第6章会再次回顾这里介绍的所有模式，并在各种语言的具体示例佐证下展开讨论。

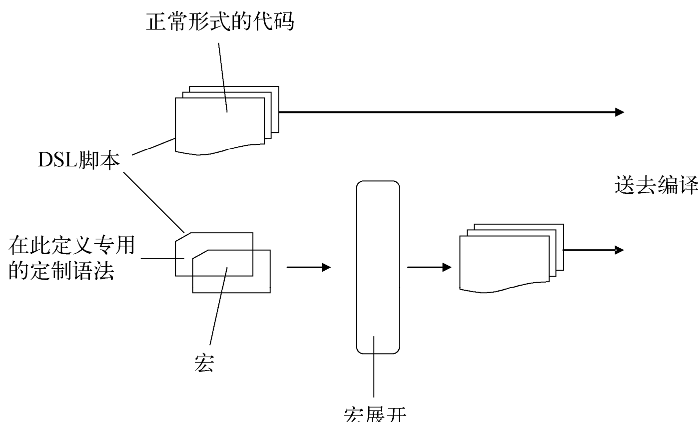


图2-7　用宏来实现编译时元编程。DSL脚本中有一部分是普通的宿主语言成分，有一部分是开发者定制的专用语法。定制部分以宏的形式出现，在宏展开阶段展开成正常的语言形式。然后，所有代码被一起送给编译器

本章一开头我们就承诺过要探讨现实中的DSL设计。前面已经讨论过用Java和Groovy完成的DSL实现，刚刚又介绍了内部DSL实现中出现的模式。每个模式都是实践者应该勇于重用的经验片段。所有这些模式都是在现实的DSL开发中成功实践过的。

说完内部DSL，显然接下来我们就要说外部DSL了。假如宿主语言无法实现你想要的DSL语法形态，那就应该跳脱宿主语言的桎梏，不惜选择需要从零开始的实现途径。外部DSL正是正确答案。接下来，我们就来看看外部DSL有哪些实现模式。

2.3.2　外部DSL模式：共性与差异性

外部DSL的设计周期和原则与通用语言设计相同。我知道这个说法肯定让你望而生畏，甚至打消你在项目中设计外部DSL的念头。这句论断在理论上完全成立，只不过DSL的语法和语义并不需要像通用编程语言那么复杂，所以其实没有那么吓人。在现实中，我们可能只需要动用正则表达式来操控一下字符串，就足以实现外部DSL的处理。不论简单还是复杂，总之所有外部DSL的共同特征就是其实现不借助宿主语言的设施。

外部DSL的处理过程可以粗略分为两个阶段，如图2-8所示。

❶**解析**　对输入的文本进行分词，通过解析器识别有效的输入。

❷**加工**　解析器识别出的有效输入在此接受处理。

如果DSL比较简单，解析器可以就地完成加工工作，那么两个阶段可以合二为一。不过，一般而言比较实际的做法是让解析器把输入文本转换成一种中间表示。根据具体情况以及DSL的复杂程度，这种中间表示可以是AST，也可以是其他更复杂的语言元模型。解析器本身的复杂程度

也不同，简单的只是字符串处理而已，复杂的也许要用到精密的语法制导翻译（syntax-directed translation）技术（这种解析技术将在第8章讨论），需要动用YACC和ANTLR等“解析器生成器”来制作。加工阶段围绕中间表示来进行，可以直接从中间表示生成目标输出，也可以将之转化为一种内部DSL，然后用宿主语言的设施处理。

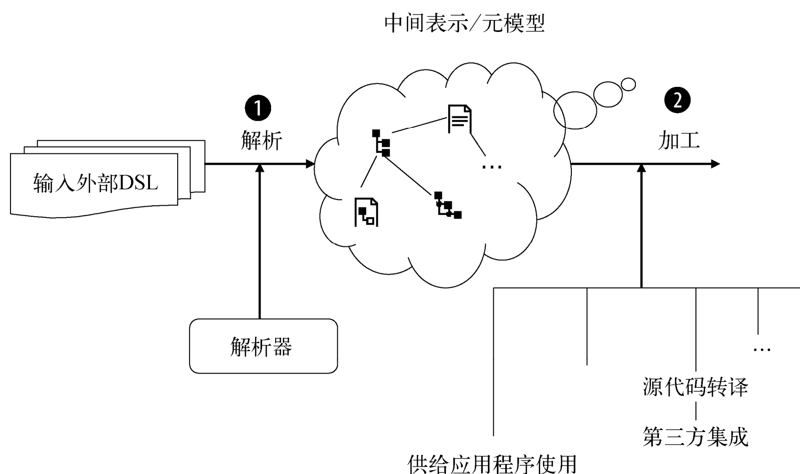


图2-8　外部DSL的处理阶段。请注意与内部DSL的区别：在这里开发者需要自行构建解析器；而对于内部DSL，解析器由宿主语言提供

接下来，我们简单看下外部DSL开发中较常遇到的几种模式。每种模式的具体实现留到第7章再作详细介绍。图2-9列举了一些现实中常见的外部DSL实现模式。

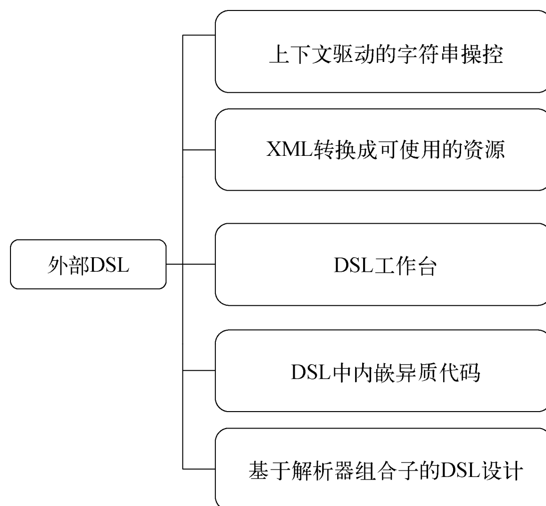


图2-9　常见的外部DSL实现模式和技巧，不太严谨的分类

图2-9中的每种模式都是一种描述DSL语法的方式，而且是不同于宿主语言的描述方式。也就是说你写下的DSL脚本，放在实现语言里面，并不是有效的语法。所以你还会看到，每种模式下产生的定制DSL语法如何被转换成为供给宿主语言使用的制品。

1. 上下文驱动的字符串操控

假设你需要处理一些业务规则，但希望向用户提供一个DSL界面来取代传统的API。考虑下面的例子（commission为“佣金”，principal为“本金”）：

```
commission of 5% on principal amount for trade
                          values greater than $1,000,000
```

这个字符串放在任何编程语言里都没有意义。但只要进行适当的“揉按拍打”，我们不难把它转换成有效的Ruby或Groovy代码。一个简单的解析器就可以完成任务，只需要做一下分词，然后套上正则表达式做简单的转换就可以了。最后得到的Ruby或Groovy代码就是可以直接执行的业务规则了。

2. XML转换成可使用的资源

大多数读者应该都用过Spring DI框架（不熟悉Spring的读者请查阅http://www.springframework.org）。其中一种配置DI容器的方式是采用一个XML配置文件，把所有依赖的抽象和实现都记入这个文件。在运行时，Spring容器载入配置文件，并将所有的依赖项关联到`BeanFactory`或者`ApplicationContext`，这两个组件将在应用程序的整个生命周期内存活，以提供所有必需的上下文信息。这个XML配置文件就是一种外部DSL，它经过解析，被持久化为可直接供应用程序使用的资源。

图2-10简单展示了Spring将XML作为外部DSL导入其`ApplicationContext`抽象的情形。

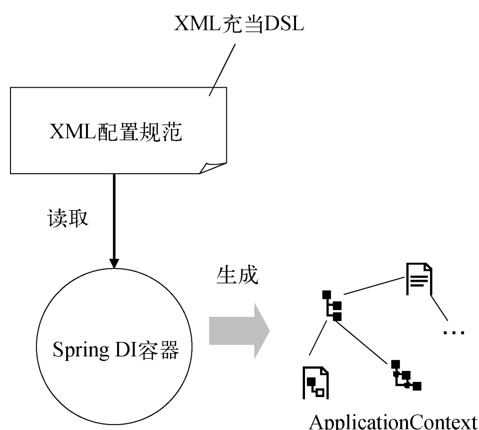


图2-10　XML被用作外部DSL，它是对Spring配置规范的抽象。容器在启动期间读入并解析XML，产生应用程序所需的`ApplicationContext`

Hibernate的映射文件是一个类似的例子，它把实体描述文件映射到数据库设计方案。（Hibernate的内容详见http://hibernate.org。）虽然两个例子的生命周期和持久化策略有所差异，但

它们都具有解析、加工两个执行阶段，这一点体现了外部DSL的共性。另一方面，这两个例子又表现出与其他模式的差异性，即其解析器的形式与复杂度、中间表示的生命周期都有别于前面讨论过的模式（上下文驱动的字符串操控）。

3. DSL工作台

我们在内部DSL的语境下讨论过元编程的核心概念，现在出现的一些语言工作台和元编程系统已经把这个概念的内涵扩展到了更高层次。编写文本形式的代码时，编译器需要解析代码并生成AST。那要是系统直接就把你写的代码以AST的形式来存放呢？如果系统能提供这样的中间表示，对其进行的转换、操控以及后续（以这种中间表示为基础）的代码生成都会简单很多。

Eclipse Xtext（http://www.eclipse.org/Xtext）是个很好的例子，它提供了开发外部DSL的全套解决方案。在这个系统中，DSL不是保存成纯文本形式，而是以元模型的形式保存DSL文法的高阶表示。这些元模型可以无缝地集成进其他框架，如代码生成器、编辑器等。像Xtext这样的工具叫做DSL工作台，因为它们提供了开发、管理、维护外部DSL的完整环境。第7章将通过详细的案例分析讲解基于Xtext的DSL设计。

JetBrains的Meta Programming System（http://www.jetbrains.com/mps/index.html）支持非文本表示的程序代码，不再需要代码解析。代码及其标注、引用总是以AST的形态存在，你只要定义合适的生成器就能生产出各种语言的代码。这就好像我们在使用工作台的元编程系统所提供的元语言设计外部DSL，用它来定义业务规则、类型、约束，跟使用平常的编程语言没什么两样。只是代码的外部表示不一样，它更友好也更容易操控，甚至可以是图形化的。

回顾图2-9，我们已经介绍了3种常用的外部DSL实现手段，还有两种在现实中特别重要的模式有待讲解。本章的第三个里程碑已经在望。对于迄今介绍的所有内部、外部DSL实现技术，相信你已经有了很好的理解，想必已经迫不及待地想看到它们在领域建模中的实际表现。请再耐心一点，你会很快就会看到相关内容。

4. DSL中内嵌异质代码

解析器生成工具如YACC和ANTLR让程序员使用类似于EBNF（Extended Backus-Naur Form，扩展巴科斯–瑙尔范式）的语法符号来定义语言的语法。生成工具“吃进”产生规则，“吐出”语言解析器。在实现解析器的时候，我们一般会定义一些操作，让解析器在识别到某些输入片段的时候执行，例如要求解析器对于输入的语言字符串生成中间表示，供应用程序在后续环节使用。

YACC和ANTLR等工具允许在生成规则中嵌入宿主语言代码编写的操作定义。最终得到的解析器代码也将每一条规则下关联的C、C++或Java片段囊括在内。在这种外部DSL设计模式下，DSL因为嵌入了别种高级语言而得到扩展。我们将在第7章把ANTLR作为解析器生成工具，按照这个模式实现一个完整的DSL设计。现在赶紧看看最后一种模式吧。

5. 基于解析器组合子的DSL设计

这是图2-9列出的最后一种模式，也是最有革新意义的外部DSL设计方法。刚刚你已经知道怎样利用YACC、ANTLR等外部工具结合内嵌编程语言来生成DSL解析器。这些工具完全胜任工作，但缺点是使用上不很友好。现在有不少语言提供了更好的替代品，也就是解析器组合子（parser combinator）。

在强大的类型系统支持下，利用解析器组合子设计而成的DSL可以实现为宿主语言的一个库。也就是说，实现过程中可以充分利用宿主语言的各种配件，例如类、方法、组合子等，不必求助于外部工具包。

Scala的标准库中已经包含了解析器组合子库。在Scala的高阶函数帮助下，我们定义的组合子可以使解析器的描述语句近似于EBNF产生规则。下面就是一段Scala解析器组合子的应用示例，它用纯Scala语言定义了一种简单的交易单处理语言的语法。

```
object OrderDSL extends StandardTokenParsers {
  lexical.delimiters ++= List("(", ")", ",")
  lexical.reserved += ("buy", "sell", "shares", "at",
    "max", "min", "for", "trading", "account")
  def instr = trans ~ account_spec
  def trans = "(" ~> repsep(trans_spec, ",") <~ ")"
  def trans_spec = buy_sell ~ buy_sell_instr
  def account_spec = "for" ~> "trading" ~> "account" ~> stringLit
  def buy_sell = ("buy" | "sell")
  def buy_sell_instr = security_spec ~ price_spec
  def security_spec = numericLit ~ ident ~ "shares"
  def price_spec = "at" ~ ("min" | "max") ~ numericLit
}
```

即使看不懂这段代码的细节部分也没关系。这所以举这个例子，我只是为了展示一下纯以宿主语言完成声明式解析器开发这一方式有多大威力。结论是纯粹用Scala开发外部DSL解析器完全可行。



第8章将再次讨论解析器组合子，其中会有通过Scala解析器组合子建立外部DSL的详细示例。

我们的讨论至此告一段落，前面列举过的所有DSL实现模式和技巧都已介绍完毕。本章的介绍都是概括性的，目的是使读者对现实中的DSL实现大环境有所了解。至于每种模式的详细用法，我们安排在第4章~第8章讲解，届时会用它们实现DSL以解决现实的领域问题。

在本章结束之前，我们最后考虑一个具有现实意义的问题，也是你每次动手设计DSL之前应该考虑的问题：怎样才能务实地决定选择哪种形式的DSL。第1章已经讨论过什么时候该用DSL，现在我要解释下该如何在内部DSL和外部DSL之间进行选择。用DSL来建模领域问题肯定没错，但同时要平衡考虑它的实现方面才好作出最终决定。无论决定设计一个内部DSL还是外部DSL，都取决于许多因素；而且并非所有因素都是技术因素。

2.4 选择 DSL 的实现方式

程序员随时都要面对许多选择，无论设计方针、编程范式，还是具体到某个实现的惯用法，都等待我们作决策。怎样选择才会有好的DSL，怎样选择才会有好的抽象，以及怎样选择才能满足领域用户对表现力的要求，这些我们全都讲过了；现在，我们要介绍一下你将会面对的另外一些选择。

当你决定让项目走上基于DSL开发的道路，也确定了能在DSL设计中派上用场的业务领域组件，这时候怎样决定实现策略呢？应该利用宿主语言把问题建模成内部DSL，还是应该为了表现力水平而选择外部DSL？这个问题和大多数的软件工程问题一样，并没有放之四海而皆准的正确答案。问题域不许做什么和解答域允许做什么，共同决定了我们的答案。在你下决心选择某种DSL实现技术之前，有几大因素是应该考虑的，本节就带你审视一番。

1. 重用现有设施

内部DSL搭了宿主语言的顺风车，所有的设施、语法、语义、模块系统、类型系统、错误报告方法、完整的工具链，内部DSL都能沾光。这一点绝对是内部DSL的最大实现优势。而对于外部DSL来说，任何设施都要从零开始建设，这绝非易事。在内部DSL范围内，我们又有多种实现模式可以选择，这在上一节都已经讨论过。决策主要取决于宿主语言的能力和它所支持的抽象层次。

如果宿主语言如Scala或Haskell那样拥有强大的类型系统，那么你可以考虑用其类型系统来表达领域类型，从而得到纯粹的内嵌式DSL。不过，类型内嵌不一定总是最优的选项，只有当客体语言的语法、语义都接近于宿主语言时，这才能取得好的效果。任何一方面不匹配都会使DSL与宿主语言的系统环境格格不入，使宿主语言的控制结构无法顺利地组织起DSL语句。在这种情况下，你也许应该求助于元编程技术——如果宿主语言提供了这种选择。前面已经介绍过，因为元编程允许扩展宿主语言，允许在其中加入原本没有的领域构造，所以最后得到的DSL表面语法能比类型内嵌方案表现力更强。

2. 充分利用现有的知识

有些时候，你只能根据团队成员现有的知识水平来选择实现范式。内部DSL在这一点上较有优势。不过要注意，程序员熟悉某种语言，并不等于就熟悉该语言下的DSL实现惯例。连贯接口在Java和Ruby中很常见，但并非没有陷阱。在具体的条件下必须考虑许多方面才能保证DSL语义上的一致性，比如抽象是否可变、连贯API是否上下文敏感，还有方法链的*收尾问题*（finishing problem，参见2.6节文献[4]）。所有这些考虑角度都牵涉到模式或惯用法的微妙变化，对DSL的一致性有不可忽视的影响。

利用现有的知识肯定是必须考虑的因素。团队领导判断成员的专业能力，不应该根据他们对语言表面语法的熟悉程度，而应该放在实现DSL的背景下去衡量。有的团队不会勉强坚持用Java实现内部DSL的方案，而是会选用XML来实现外部DSL，并且极大地提高了生产效率，也赢得了用户的认可。

3. 外部DSL的学习曲线

也许你不敢选择外部DSL，因为觉得设计外部DSL就像设计通用编程语言那么复杂。我不怪你有这样的想法。只要一提*语法制导翻译*、*递归下降解析*、LALR和SLR这些术语，人们就很难不鉴于复杂性被它们吓到。

现实中，大多数应用程序所要求的外部DSL没必要做得像完整的编程语言那么复杂。不过，确实有些外部DSL相当复杂，必须将其学习曲线纳入开发成本去考虑。外部DSL的优点是可以定制几乎任何东西，连如何处理错误和异常都可以定制，你不会因为宿主语言的限制而被束手束脚。

4. 恰当的表现力水平

虽然内部DSL对现有设施的重用是很大的优势，但宿主语言强加的约束使你设计出来的DSL很难达到领域用户要求的表现力水平，这也是事实。在现实中，往往要等到开发环境和工具链都已经不可能更改的时候，我们才发现有的模块很适合应用DSL。因此，你不见得有机会选择最合适的语言来设计内部DSL。

在这种时候，我们有必要考虑在应用程序架构中纳入外部DSL。用外部DSL建模领域问题的最大优点，是你可以把语言的复杂度设计得正好和手头问题相匹配。外部DSL给予开发者充足的调整空间去适应用户反馈，而内部DSL就不一定能做到这一点，因为语法、语义始终在宿主语言的约束之下。

5. 组合性

在典型的应用程序开发场景中，不同的DSL或者DSL和宿主语言都有可能需要组合起来使用。内部DSL与宿主语言的组合很简单，毕竟DSL是使用同种语言实现的，而且一般都实现成宿主语言的一个库。

组合使用多种DSL就值得讨论一番了，即使所有DSL的宿主语言都一样，情况也不那么单纯。对于静态类型语言下实现的几种内嵌式DSL，必须在宿主语言类型系统的支持下才有可能无缝地组合在一起。支持函数式编程范式的语言，一般鼓励你基于函数式的组合子设计内部DSL。如果设计得当，内部DSL和组合子完全可以组合。外部DSL比较难做到组合使用，尤其当两种DSL被分别设计，又没有预先考虑组合要求的情况下，就更不可能实现了。

2.5　小结

从第1章的基本原理，到本章实用主义的DSL用法、实现、分类，你已经在很短的时间内汲取了大量知识。如果说第1章只是指引你步入DSL开发，那么本章就切切实实地把你带到了DSL的现实世界。

本章一开头就举例强调基于DSL的程序开发重点是抽象的表现力。Java实现的交易单处理DSL对于作为用户的程序员来说表现力已经足够。但如果打算让不懂编程的领域专家用你的DSL作为表达载体，你就要有一种更能传达领域含义的实现语言。Groovy实现做到了，从Java迁移到Groovy大大提高了其表现力水平。

接着，我们的话题从具体实现转为更广泛的DSL模式。你了解到在内部和外部DSL两大分类下还有很多不同的实现模式。DSL的复杂度各不相同。DSL设计者需要挑选最适合手头问题的实现策略。然后，2.3节逐一介绍各种模式，以便你对现有的DSL实现技术有个概括的了解。

阅读完本章，你已经见识过五花八门的DSL形式和结构了。最终把领域模型塑造成什么样，这是架构师的责任。不过在继续建模的话题之前，我们需要先解决DSL与开发环境的集成问题。目前为止，我们已经见过DSL的宏观模型发挥作用。现在请换一种思维，想一想DSL开发中的微观建模产物。假设你在JVM平台上用Java开发了核心应用程序，然后又开发了Groovy DSL，请问两方面要怎样集成，才能保证DSL既可以访问Java组件，又能维持其独立实体的身份？为了回答

这个问题，你要面对许多选择和陷阱；这些都是下一章的内容。

要点与最佳实践

- **Java的语言特性足以实现出具有充分表现力的DSL**。如果你感觉Java有所局限，请关注JVM平台上与Java互操作性良好的其他语言。
- **当你为DSL选定一种实现语言，请注意学习那些使DSL贴近实现语言习惯的模式**。当选用Groovy或Ruby时，元编程模式是最好的帮手。当选用Scala时，它强大的类型系统可以成为DSL实现的靠山。
- **选择DSL实现形式的时候，要保持开放的心态**。外部DSL看似困难，但实践中很少有必要从头实现一种完整全面的语言，所以复杂度不一定有那么高。

在后面的章节里，我们将要实现证券交易领域的各种DSL片段，老鲍当然也会陪着我们，用他洞悉全局的眼光帮忙改进DSL的表现力。

2.6　参考文献

[1] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[2] Hudak P. 1998. Modular Domain-Specific Languages and Tools, *Proceedings of the 5th International Conference on Software Reuse*.

[3] Hofer, Christian, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic Embedding of DSLs, *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pp 137-148.

[4] Ford, Neal, *Advanced DSLs in Ruby*, http://github.com/nealford/presentations/tree/master.

第 3 章

DSL驱动的应用程序开发

本章内容

- ❑ 将内部和外部DSL集成进核心应用程序
- ❑ 管理错误和异常
- ❑ 优化性能表现

前两章我们在用户和实现的层次对DSL进行了多方位的观察和探讨。可以看到，提高抽象的表现力可使代码更易于理解，可以帮助缩短与领域专家之间的反馈循环。不过，不管DSL设计得多么出色，归根结底你要把它和核心的Java应用程序模型集成在一起。（倒不一定是Java应用程序，这里只是举例。）集成牵涉到的众多方面也需要你未雨绸缪。以上种种，以及用DSL来开发一个完整应用程序会遇到的其他问题，就是本章的讨论内容。

一般来说，我们会用平台上的主要语言（比如Java）来开发应用程序的主体部分。然后，对于一些业务规则或者配置规范，我们可以用比Java表现力更强的语言写成DSL来表述。那么，DSL部分应该怎样跟核心应用程序无缝集成呢？DSL的演变往往独立于应用程序的主体部分，所以架构上要足够灵活，不能妨碍DSL时时的改变，还要使变化对应用程序主体的影响尽可能小。本章对以上问题的讲解计划参见图3-1。

关于DSL驱动的应用程序开发，我们主要探讨3个方面：

- ❑ 集成问题；
- ❑ 处理异常和错误；
- ❑ 管理性能表现。

DSL不能独立发挥作用。集成DSL到应用程序需要考虑诸多问题，其中有一项是错误处理：无论DSL还是核心应用程序都有可能产生异常。你打算怎样向DSL用户报告错误？又打算怎样处理？（我们将在3.4节尝试解答。）DSL使用中还可能出现各种性能问题，本章结尾部分对此有个简要的总结。

读完本章，你将学会安排应用程序的架构，使之能与别种语言编写的DSL无缝集成。

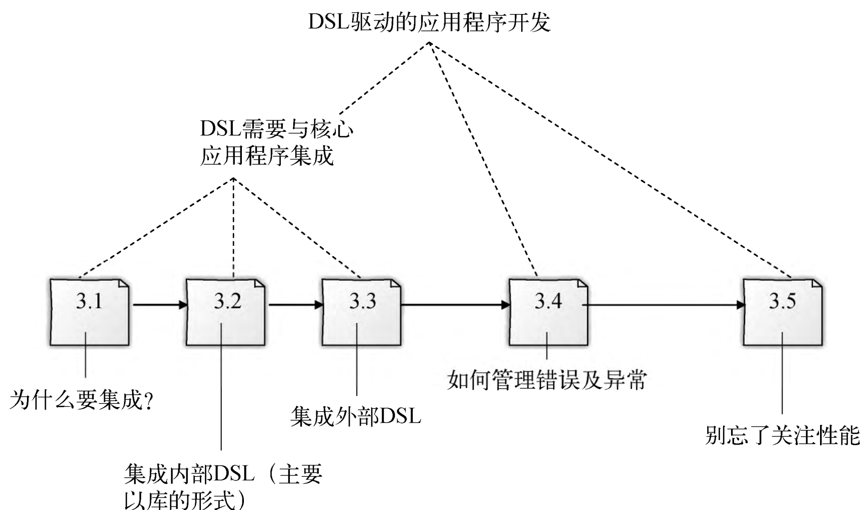


图3-1 了解本章学习计划，学习DSL驱动应用程序开发的各种问题

3.1 探索 DSL 集成

DSL是一种优美的抽象，它也毫不例外地需要与应用程序架构中的其他组件集成。在一般的使用场景中，DSL所建模的制品属于应用程序中易变的部分，比如业务规则和配置参数。所以DSL和应用程序主体要能够以不同的步调各自独立地演变，同时它们又能够与工作流无缝地结合，这两点都是非常重要的设计要求。

这一节，我们将探索无缝集成DSL的各种途径。一种DSL本身只针对一个专门的领域，却有可能在多个更大的领域中使用。至于实际上是否被广泛使用，还要看它所针对的领域的通用程度。例如，一种处理日期时间的DSL在任何需要计算日期的程序中都用得上，而一种针对企业税收规定的DSL用途就非常有限了。另外，因为日期DSL要考虑集成到多个应用程序上下文的情况，所以它要具有更强的适应性。

我们稍后便深入介绍DSL集成问题，在此之前，请看图3-2，一个DSL驱动的应用程序架构大体上就是这个样子。

对于典型的多层架构，DSL可以用在任何一层，只要它准备好该层要求的集成上下文信息就行。集成内部DSL相对容易，因为内部DSL一般与应用程序使用同种语言，且被设计为库的形式。集成外部DSL较为麻烦，需要建立某种插入机制，对外公开专门的端口给应用程序对接。我们不急着列举内内部和外部DSL集成到应用程序架构的具体用例，先来谈谈为什么需要小心处理DSL的集成问题。

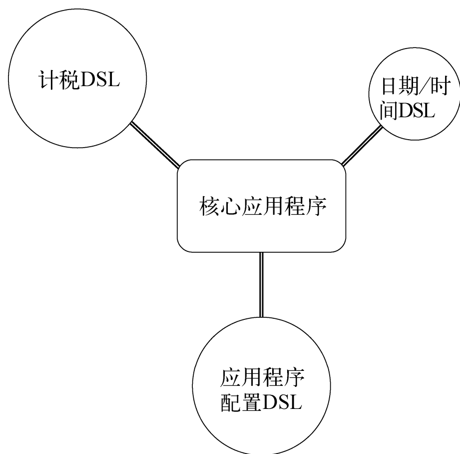


图3-2 宏观视角下基于DSL的应用程序架构，注意DSL与核心应用程序的解耦情形。不同部分的演变时间表各不相同

为什么关心DSL集成

核心应用程序对内有各种组件要连接，对外有DSL脚本要插入，两者都需要你小心处理。DSL的演变步调独立于核心应用程序，所以两者之间的耦合程度要恰如其分。



如果选用Groovy、Ruby、Scala等表现力强的语言作为核心应用程序的主要语言，基本上不存在什么集成问题；根本就没必要插入其他语言的DSL脚本。所以，接下来的内容主要与在Java应用程序中集成DSL脚本的情况有关。

开发者很容易鲁莽地决定在一个应用程序内使用多种语言的DSL，却忘了想想到时候要怎么集成。如果选错了DSL的实现语言，图3-2中好端端的架构可能会变成开发者的噩梦。没有人希望落到好像图3-3所示那般境地。

如果想做到DSL与应用程序无缝集成，不想陷入图3-3中那位仁兄的境况，那么你就必须好好考虑表3-1列举的几个问题。

表3-1 将DSL集成到核心应用程序

待解问题	架构师应做事项
关注点分离	
一方面是DSL要解决的核心问题，一方面是DSL要应付的应用程序上下文，怎样保证两者之间泾渭分明？	正确定义DSL的适用上下文，并考虑DSL在当前应用之外可能的应用场景 参考附录A中作为一个核心特质探讨的抽象设计原则：精炼
DSL API的演化	
DSL API的演变要独立于应用程序上下文	确保API在演变过程中维持向后兼容 如果使用第三方提供的DSL，一旦发现它的API引入了不兼容的改动，就要立即警觉起来，否则说不定什么时候会被“反咬上一口”

（续）

待解问题	架构师应做事项
避免语言摩擦	
各种DSL所用的语言太多，会给整个架构的开发和维护造成混乱	确保DSL的实现语言可以跟应用程序的宿主语言无缝地互操作
这不仅是技术问题，还是人员问题。当程序员被迫维护太多种语言写成的代码，他们会失去合作的意愿	宁可部分牺牲DSL语法的灵活性，也不应选择无法很好地集成到核心应用程序的语言。不能因为不同语言运行在相同虚拟机（VM）上就认为其间能无缝地互操作，请记住这个提醒
	如果DSL的实现语言可以通过多种途径与应用程序的宿主语言互操作，请优先选择具有最自然集成方式的一那种；基于沙盒的脚本环境虽然是一种通用的集成途径，只宜作为后备选项。我们将在3.2节以Groovy和Java互操作为例来说明各种集成方式

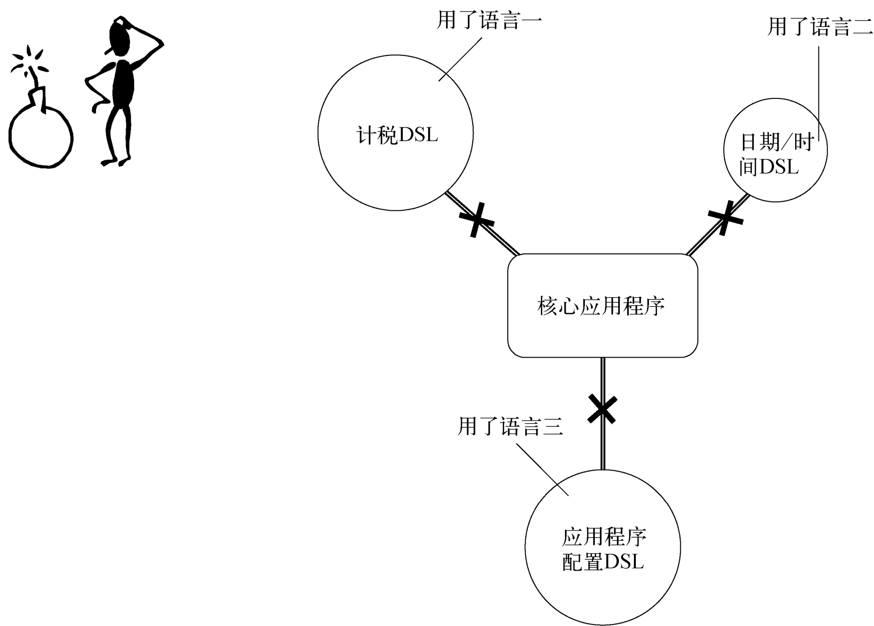


图3-3　架构师如坐针毡，苦思冥想怎样把不同语言写成的DSL跟核心应用程序集成。你能帮他解除这个定时炸弹吗

你已经知道为DSL和核心应用程序制定集成策略的必要性，接下来该了解一下各种策略的使用模式了。首先说明的是内部DSL的集成模式，内部DSL主要做成库的形式，以库中API的形式进行集成。

3.2　内部 DSL 的集成模式

设计成库形式的内部DSL，其实现语言要么和应用程序主体相同，要么可以与之无缝互操作。不管哪种情况，集成都无需借助任何外部设施来完成；无非是在DSL与核心应用程序之间发生的API调用而已。因为在同一的VM约束下，各种语言的互操作融洽无间，我把这种情况称为同质集成。请看图3-4，可以看到分别用Java、Groovy和Spring配置语言实现的不同DSL可无差别地集成于JVM之上。每种DSL都可以做成jar文件来部署，主体应用程序只需引用jar文件即可。

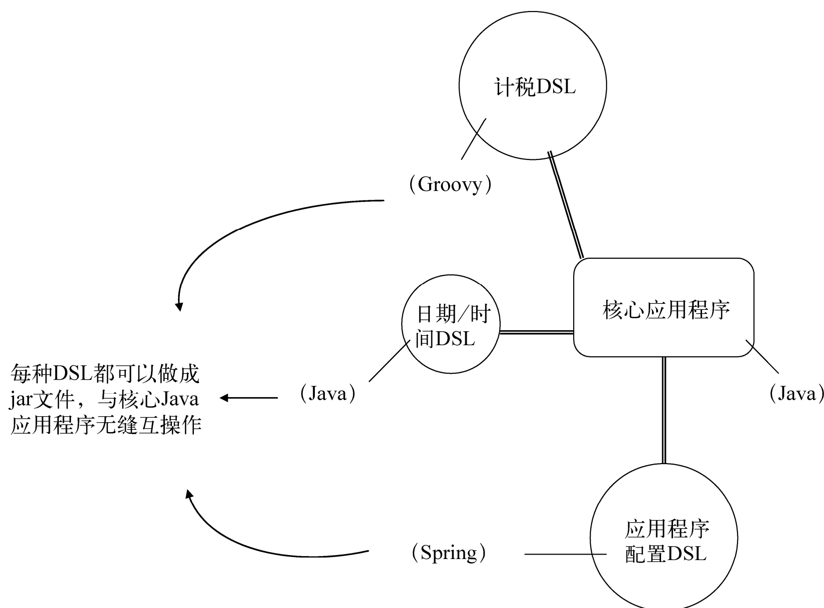


图3-4　3种DSL均与核心应用程序同质集成。每种DSL都可部署为jar文件，以便与核心Java应用程序在JVM上无缝地互操作

假设你主要用编程语言Java开发应用程序，但因为自己有多语言能力，所以打算利用Groovy的优势实现XML Builder功能。（这种Builder是在Groovy下处理XML的绝佳方式，参见http://www.ibm.com/developerworks/java/library/j-pg04125/。）没多久你又发现，有个第三方的JRuby DSL很适合用来加载Spring bean以管理应用程序配置。这时候，应该怎样把好几种DSL和核心应用程序集成在一起呢？集成不可以让用户面临太多复杂性；同时，DSL要与核心应用程序保持足够的独立性，以便独立掌握其演变步调和生命周期。JVM语言所写的DSL可以通过多种方式与Java应用程序集成。

众多JVM语言大都提供了与Java集成的多种途径。表3-2列出的每一种方式都能达到集成目的，但你必需了解每一种方式的优缺点，从中选出最适合的。

表3-2　内部DSL的集成入口

内部DSL的集成模式	集成入口
Java 6的脚本引擎（参见3.2.1节）	用Groovy等脚本语言编写的DSL可通过Java 6提供的相应脚本引擎来集成
DSL包装器（参见3.2.2节）	利用JRuby、Scala、Groovy等语言把Java对象包装成更灵巧的API，这些语言本身就有Java集成功能
语言特有的集成功能（参见3.2.3节）	通过实现一种直接加载并解析DSL脚本的程序抽象直接与Java集成。Groovy具有这样的直接集成能力
基于Spring的集成（参见3.2.4节）	通过Spring的声明式配置直接加载用动态语言编写的各种bean到应用程序



接下来讨论集成的时候，我们都假定核心应用程序是以Java语言开发的，这种情况当今最为普遍。另外请注意，虽然我们提到的几种语言全都具备不同程度的Java集成能力，但它们彼此之间的集成还不成熟。现在还没有人在Groovy应用程序里面内嵌Ruby写的DSL。

下面我们就来一一详述表3-2所列的模式，看看一些JVM语言是怎么利用它们集成到Java应用程序的。

3.2.1　通过Java 6 的脚本引擎进行集成

Java平台普及程度非常高，所以早有人考虑为Java平台上的所有语言建立统一的互操作平台。Java 6的脚本特性允许通过相应的引擎在Java应用程序中嵌入脚本语言。通过`javax.script`包内定义的API，完全可以用于嵌入Groovy、JRuby等语言实现的DSL。来看这种集成方式的一个示例，其中还是沿用第2章中的交易单处理DSL。

1. 准备Groovy DSL

在2.2.2节，我们写了一段Groovy脚本来执行创建订单的DSL。现在还是同样的DSL，但这次要在Java应用程序里面集成和调用。这个例子将让你认识Java脚本特性开启DSL集成通道的能力。

代码清单3-1中就是我们用来处理客户交易单的DSL的Groovy实现（`ClientOrder.groovy`，其内容与2.2.2节中的相同）。

代码清单3-1　`ClientOrder.groovy`：Groovy语言编写的交易单处理DSL

```
ExpandoMetaClass.enableGlobally()

class Order {
  def security
  def quantity
  def limitPrice
  def allOrNone
  def value
  def bs

  def buy(su, closure) {
    bs = 'Bought'
    buy_sell(su, closure)
```

```
    }

    def sell(su, closure) {
      bs = 'Sold'
      buy_sell(su, closure)
    }

    def getTo() {
      this
    }

    private buy_sell(su, closure) {
      security = su[0]
      quantity = su[1]
      closure()
    }
  }

  def methodMissing(String name, args) {
    order.metaClass.getMetaProperty(name).setProperty(order, args)
  }

  def getNewOrder() {
    order = new Order()
  }

  def valueAs(closure) {
    order.value = closure(order.quantity, order.limitPrice[0])
    order
  }

  Integer.metaClass.getShares = { -> delegate }
  Integer.metaClass.of = { instrument ->  [instrument, delegate] }
```

我们在上面的Groovy代码中实现了一个`Order`抽象来反映用户输入的交易单详情。而在另一个脚本文件order.dsl（代码清单3-2）中，DSL用户利用代码清单3-1中的实现把用户下单的操作写成脚本——这个脚本也是Groovy代码。注意，用户脚本完全基于我们在代码清单3-1中设计的DSL，用户只需要具备最低限度的编程知识。除了建立交易单，脚本还将交易单收集到一个集合，然后返回给调用者。但调用者是谁呢？别急，你马上就会知道。

代码清单3-2　order.dsl：执行下单操作的一段Groovy脚本

```
orders = []
newOrder.to.buy(100.shares.of('IBM')) {
  limitPrice   300
  allOrNone    true
  valueAs      {qty, unitPrice -> qty * unitPrice - 500}
}
orders << order

newOrder.to.buy(150.shares.of('GOOG')) {
  limitPrice   300
  allOrNone    true
  valueAs      {qty, unitPrice -> qty * unitPrice - 500}
}
```


❶ 将交易单加入集合

```
orders << order

newOrder.to.buy(200.shares.of('MSOFT')) {
  limitPrice   300
  allOrNone    true
  valueAs      {qty, unitPrice -> qty * unitPrice - 500}
}
orders << order
orders
```

❷ 将集合返回给调用者

在代码清单3-2中，用户写下`newOrder`建立一个新的`Order`抽象，然后填入各种属性，比如买入卖出、交易数量、限价、定价策略等。所有新建立的交易单都被放入一个集合❶，该集合在在❷的位置被返回。

至此铺垫工作都已完成，重头戏就要上场了：我们要把DSL实现和用户脚本都集成到Java应用程序主体。

2. 集成DSL实现及用户脚本

代码清单3-3是从应用程序主体中截取的代码片段，它等着DSL脚本执行后返回的一个交易单集合，然后对交易单进行后续处理。这里用到的脚本引擎是Groovy语言的；还有JRuby、Clojure、Rhino和Jyphon等其他JVM语言的引擎，也可以像Groovy的一样无缝整合到Java应用程序里（详见https://scripting.dev.java.net/）。

代码清单3-3 调用Groovy DSL的Java程序代码

```
ScriptEngineManager factory = new ScriptEngineManager();
ScriptEngine engine = factory.getEngineByName("groovy");

List<?> orders  = (List<?>)
  engine.eval(new InputStreamReader(
    new BufferedInputStream(
      new SequenceInputStream(
        new FileInputStream("ClientOrder.groovy"),
        new FileInputStream("order.dsl")))));

System.out.println(orders.size());
for(Object o : orders) {
  System.out.println(o);
}
```

❶ 取得脚本引擎的工厂
❷ 取得Groovy脚本引擎
❸ 返回交易单的列表
❹ 执行DSL脚本
❺ 处理交易单

对照代码清单3-3和图3-5，我们不难看清集成的每一个步骤。图3-5的顺序图上标注了代码清单3-3中对DSL脚本及实现所进行的操作。

显然，Java 6的脚本API几乎能够集成任何JVM语言编写的DSL到Java应用程序。`javax.script`包内的API还能用于设置各种作用域的变量绑定，以便在DSL与Java组件之间交换信息。

3. Java 6脚本特性的不足

Java 6脚本特性是实现JVM语言互操作的一种极通用方式。但有所谓通用策略，就说明有专门针对某种语言的更好选择。由于DSL脚本被一个单独的`ClassLoader`加载，又在独立的沙盒中运行，Groovy抽象与Java抽象之间存在互操作问题。注意，在代码清单3-3中，Groovy DSL脚本返回的`Order`列表，到了Java一侧就成了`Object`列表。要在这些对象上调用`Order`抽象定义的方

法，只好利用反射。另外，由于脚本在ScriptEngine的沙盒中执行，当出现异常时，栈跟踪信息中显示的行号无法对应到源文件中的行号。因此，DSL脚本抛出的异常调试起来比较困难。因为Java 6脚本这样那样的不足，我们有必要继续探索更好的内部DSL集成方案。

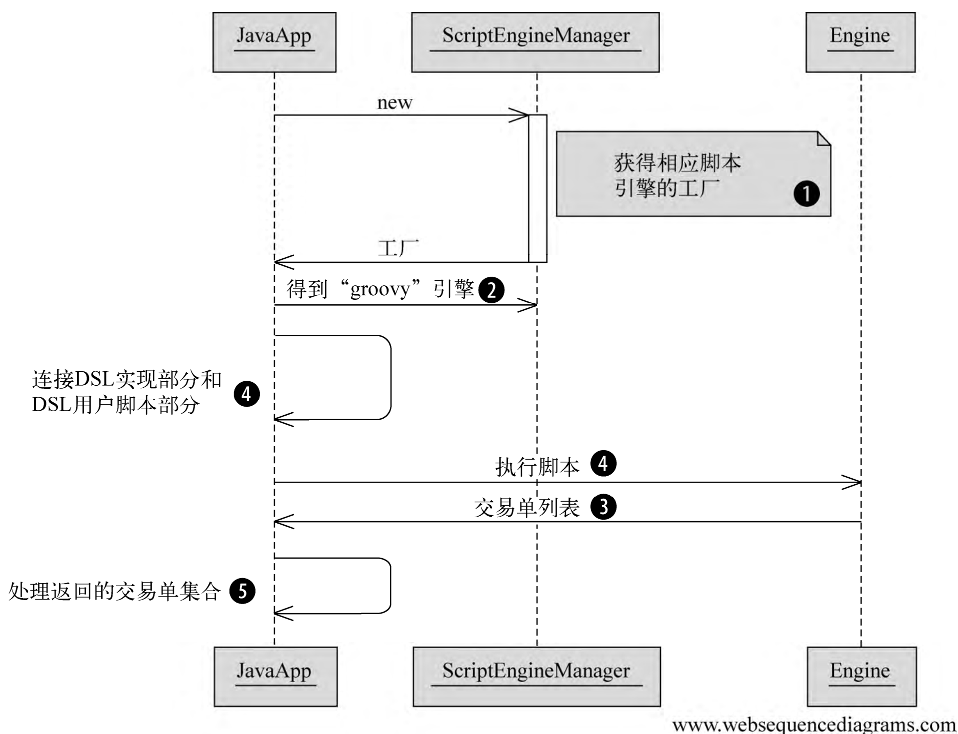


图3-5 通过Java 6的脚本引擎集成Groovy DSL。这幅交互图反映了在ScriptEngine的沙盒内执行Groovy DSL脚本的全部步骤



脚本引擎是从Java 6开始引入的，是一种在Java程序内部执行脚本的通用方法。按照ScriptEngine相关API的设计原则，任何JVM语言只要实现了JSR 233规范要求的设施，就能获得该特性的支持。如果你打算用于实现DSL的语言有专门的Java集成途径，应该优先考虑，仅将JSR 233兼容的方案作为退而求其次的选择。语言特有的方案一般较为简单，也更符合语言习惯，所以往往效果最好。

Java 6的脚本API成就了JVM上多语言并用的现象。我们举了Groovy的例子，这纯粹是因为第2章刚好实现了一段Groovy DSL，它很容易无缝插入到Java应用程序。同样的集成手段完全适用于其他JVM语言（例如JRuby、Clojure和Rhino）编写的DSL。



即使在单个解答域中，多语言并用现象也鼓励使用多种语言。并用的语言需要有良好的互操作性，还要有明晰的集成入口。通常并用的语言享有共同的运行时平

台，如JVM，其上的语言有Java、Scala、Ruby、Groovy等。DSL一个根本的设计思路就是选择最适合的语言设计领域API，然后通过共同的运行时平台将之与核心应用程序集成在一起。

DSL可以集成到不同层次。3.2.1节中探讨的Java脚本方案允许将DSL嵌入到ScriptEngine执行框架，然后调用DSL脚本。它的优点是DSL完全与应用主体解耦，在ScriptEngine的沙盒中执行；缺点是DSL组件不容易和应用程序的主体环境交互，操作不直观。

下面我们来看另一种DSL集成方式，它的工作层次不同于脚本引擎，而且与应用程序的宿主语言结合得更为紧密。

3.2.2　通过DSL包装器集成

在这种集成方式下，我们利用其宿主语言的丰富特性，将DSL构建成主体应用程序组件外面的包装层。遗留系统很适合采用这种方式将其对外接口改造成更灵巧的API。JVM语言大多提供了相当丰富的语言特性，完全有能力将遗留抽象"装饰"成更具表现力的领域组件。对于改造结果，不仅领域用户会很满意，而且开发者中的API用户也会乐见其成。

1. 示例

这次的示例中我们使用Scala语言，它是JVM上的静态类型语言，有完善的Java互操作能力。假设你的应用程序主体是用Java写成的，而且所有领域对象都已包含其中。这时客户因为受了"蛊惑"想试试基于DSL的开发，所以要求你在现有的Java交易系统上增加一些光鲜的DSL特性。这种场景正好适合包装式集成。

为了便于解释包装式集成概念，我会采用另一个证券交易的例子进行讲解。图3-6大体展示了交易过程。别忘了看一下补充内容"金融中介系统：客户账户"，你需要知道里面介绍的背景信息。

 **金融中介系统：客户账户**

为了完成交易，客户需要在STO（Stock Trading Organization，证券交易组织）开设一个账户（称为**交易账户**）。客户参与的所有交易都被记入这个账户，并在STO留有交易记录。一旦成交，结算过程就必须启动。结算过程核算出交易双方各自结余的证券和现金数量。

来看例子。客户XXX通过STO野村证券买入100股索尼的股票，每股为50美元。STO从交易所获得某中介卖出的证券。成交后有个结算过程，交易双方在此过程中互换100股索尼股票和约5000美元。结算在一个账户上进行（称为**结算账户**），这个账户可以和客户的交易账户相同，也可以是另一个账户。

总之，交易账户用来交易，结算账户用来结算交易。两个账户可以相同也可以不同。图3-6是交易、结算过程的概略图。

来看Account（账户）领域模型。账户是证券交易领域的一个实体，券商、客户、中介都通过它来管理交易和结算活动。刚刚的补充内容简单介绍了交易和结算操作中出现的不同账户类型和它们的功用。

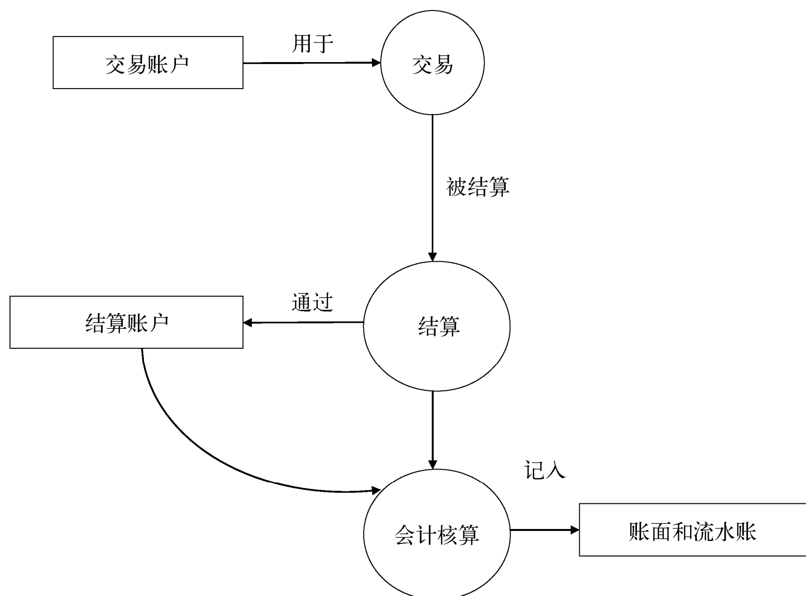


图3-6　交易账户和结算账户在交易过程中的作用



要是记不清楚“交易”和“结算”在此领域中的确切含义，请回顾第1章和第2章的补充内容。

代码清单3-4就是Account实体的领域模型，我们用它来讨论包装式DSL集成。代码中略为删减了一些不重要的细节。我们将要实现的Scala包装器就建立在Account这个Java类基础之上。看过这个例子，你将看到包装器模式对客户API的改造成效，由API组织起来的语句结构将更为精干和具有表现力。

代码清单3-4　Java语言编写的Account领域模型

```
public class Account {
    public enum STATUS { OPEN, CLOSED }

    public enum TYPE { TRADING, SETTLEMENT, BOTH }

    private String number;
    private String firstName;
    private List<String> names = new ArrayList<String>();
    private STATUS status = STATUS.OPEN;
    private TYPE accountType = TYPE.TRADING;
    private double interestAccrued = 0.0;

    public Account(String number, String firstName) {
        this.number = number;
        this.firstName = firstName;
    }
```

```
    public Account(String number, String firstName, TYPE accountType) {
        this(number, firstName);
        this.accountType = accountType;
    }

    //省略了获取方法

    public double calculate(final Calculator c) {
        interestAccrued = c.calculate(this);
        return interestAccrued;
    }

    public boolean isOpen() {
        return status.equals(STATUS.OPEN);
    }

    public Account addName(String name) {
        names.add(name);
        return this;
    }
}
```

如果你曾经做过Java编程，肯定已经对代码清单3-4所示模型中的种种累赘、死板的内容见怪不怪了。让我们试着把抽象变得更灵巧一点，给客户准备一套领域味道更浓、表达更自如的API。到时候，我们的DSL和Java应用程序将真正血脉相连，犹如一体。

2. 建造DSL

首先，我们建立一个Scala抽象AccountDSL充当Java类Account的适配器，实现所谓的灵巧领域API。我们必须给Account类披上极其灵巧的外衣，不管DSL如何设计，让用户都可以把DSL用于现有的Account类实例；这是最终目标。后续几段代码将向你演示如何逐步强化AccountDSL抽象。随后我们还会讨论可能出现的DSL使用情形，让你感受一下领域抽象的强化成果。

代码清单3-5是用Scala编写的DSL层，它将与Java类Account无缝集成使用。

代码清单3-5　用Scala语言编写的AccountDSL

```
class AccountDSL(value: Account) {

  import scala.collection.JavaConversions._
  def names =                                                    ❶ 将Java集合转换为Scala集合
    value.getNames.toSeq.toList ::: List(value.getFirstName)

  def belongsTo(name: String) = {                                ❷ 领域API
    (name == value.getFirstName) || (names exists(_ == name))
  }

  def <<(name: String) = {                                       ❸ 作用于集合的新运算符
    value.addName(name)
    this
  }
  //..
}
```

代码中用到一些典型的Scala惯用法，请参阅补充内容“Scala基础知识”里的简要介绍。欲详细了解Scala知识，请参阅3.7节文献[1]。

Scala基础知识

在belongsTo方法里面，我们写了一句断言：

```
>> (names exists(_ == name))
```

它其实是一种简略写法，意思等同于以下Scala代码：

```
>> (names.exists(n => n == name))
```

(1) 在Scala语言里，调用方法的时候方法和接收者之间的点符号（.）是可选的。

(2) 在Scala语言里，下划线符号（_）可以作为简写记号代表别的东西。代码清单3-5中的_是当做占位符使用的，提供参数给exists所接受的高阶函数。

(3) exists所接受函数的参数类型由Scala类型推断器进行推断。

(4) 在Scala语言里，运算符也是方法。我们可以定义一个给账户对象添加客户名字的<<方法。有些人觉得像<<这样的运算符号比较好看，但我必须提醒一下，好看与否完全属于个人喜好，如果用得太多反而可能降低代码的可读性。

代码清单3-5中的AccountDSL是Java类Account的适配器，Account被包裹起来，成为了AccountDSL的底层实现。在❶的位置，我们为了方便后面的高阶函数，把Java集合转换成了Scala集合。（Scala集合上可以施用高阶函数，所以更能表达清楚一些操作的意思，在这个意义上说，Scala集合的语义总是比Java集合更丰富。）这里用到了Scala 2.8才有的Java、Scala集合隐式（implicit）转换功能，如果你用的Scala版本比较低，可以改用jcl转换API：

```
def names =
  (new BufferWrapper[String] {
    def underlying = value.getNames
  }).toList ::: List(value.getFirstName)
```

我们还定义了一个领域API belongsTo❷，其中用到了高阶函数和刚刚转换而来的Scala集合。这个地方充分体现了Scala紧凑的特点。最后，我们为了DSL的表现力和简洁性，特意定义出像<<这样类似运算符号的语法❸。

新的Scala API包装了原来的Java实现，不久我们也将见识到客户怎样用新的简洁语法表达其领域意图。表现力和简洁性是DSL驱动开发的主要优点，我们的例子清楚地展现了DSL的力量。

3. 利用Scala的隐式特性

在将视角转向客户之前，我们还有一件事情需要解释。只有Account和AccountDSL之间能互操作，AccountDSL上面建立的各种机巧才能应用到Account实例。通过Scala的隐式特性，我们可以让AccountDSL具备的任何功能同时对Account类的实例生效。我们所要做的，只是在词法作用域内定义一个隐式转换：

```
implicit def enrichAccount(acc: Account): AccountDSL =
  new AccountDSL(acc)
```

这样就可以透明地从Account转换成AccountDSL，之后你就可以在Account实例上使用新的DSL API了。现在，我们创建几个Java类Account的实例：

```
val acc1 = new Account("acc-1", "David P.")
val acc2 = new Account("acc-2", "John S.")
val acc3 = new Account("acc-3", "Fried T.")
```

Scala的隐式特性

enrichAccount方法定义前面有个implicit修饰符。在Scala语言中，implicit修饰符用于方法表示定义从一个类型到另一个类型的自动转换。在这里，enrichAccount方法将一个Account实例转换成AccountDSL实例。使用中并不需要写成：

```
scala> enrichAccount(acc1) belongsTo("David P."),
```

你可以直接在Account实例上调用AccountDSL的方法：

```
scala> acc1 belongsTo("David P.")
```

就好像AccountDSL的全部方法都注入到了Account类一样。是不是有点象Ruby的猴子补丁（monkey patching）？我们可以用Ruby的猴子补丁打开任意类，并向里面添加方法。

不过Scala的情况有些不一样：implicit被限制了词法作用域。Account和AccountDSL之间的自动转换只存在于enrichAccount方法的词法作用域内。而Ruby的开放类允许在全局作用域内修改现有类，这是很大的不同。3.7节的文献[3]深入分析了Scala语言中隐式特性的优点。

现在，我们使用新的<<运算符把将几个账户所有人的名字加到账户acc1：

```
acc1 << "Mary R." << "Shawn P." << "John S."
```

留意上面代码片段表现出来的言简意赅的特点，同样的意思如果用原本的Java API写出来是这样的：

```
acc1.addName("Mary R.").addName("Shawn P.").addName("John S.");
```

接着我们把几个账户组成集合，对于账户所有人中包含“John S.”的，都打印出账户所有人的名字（firstName）：

```
val accounts = List(acc1, acc2, acc3)
accounts filter(_ belongsTo "John S.") map(_ getFirstName) foreach(println)
```

表现力真好，跟原本使用Java API时简直不可同日而语。这么大的区别应该归功于Scala语言的丰富表达手段，它使我们能用一组更紧凑的API表达出更充沛的语义。上面的代码片段运用filter、map、foreach组合子来操作高阶函数，比命令式的Java语法利落得多。有没有感觉到一点兴奋？我们继续吧！

取得属于John S.的账户列表，对于其中累计利息已超过阈值（threshold）的所有账户合计其应付利息（accruedInterest）：

```
accounts.filter(_ belongsTo "John S.")
  .map(_.calculate(new CalculatorImpl))
.filter(_ > threshold)
.foldLeft(0.0)(_ + _)
```

上面的代码片段运用了之前补充内容中讲解过的一种Scala惯用法。filter后面的断言有个需要推断类型的参数，_就是代表这个参数的占位符。类似Java那样语法烦琐的语言就没办法把领域问题表达得像这里一样清楚。第1章就说过，这一切都归功于Scala等更强大语言丰富的抽象设计能力，它们减少了代码中的非本质复杂性（accidental complexity）。

注意作为calculate()方法输入使用的CalculatorImpl对象。Calculator是在Java中定

义的接口，而CalculatorImpl是其实现：

```
public interface Calculator {
    double calculate(Account account);
}

public class CalculatorImpl implements Calculator {

    @Override
    public double calculate(Account account) {
        //具体实现
    }
}
```

大多数时候，传递给Account#calculate()方法的参数总是Calculator接口的同一个实现，这时可以利用DI在运行时动态注入实现以避免代码重复。不过，Scala可以提供更好的办法：把这一参数隐含在所有的calculate调用中。

```
class AccountDSL(value: Account) {

//同上一段

  def calculateInterest(
    implicit calc: Calculator): Double = {        <-- ❶ 隐含的Calculator实例
    value.calculate(calc)
  }
}
```

上面的代码给calculateInterest方法定义了一个implicit参数，并且在DSL的执行作用域内设置该implicit参数的默认值❶。现在，我们给Calculator规定了一个隐含的默认实现，不再需要每次调用calculateInterest方法都传递一个Calculator实例。最后，计算John S.名下所有账户的应付利息就变成这个样子：

```
implicit val calc = new CalculatorImpl

accounts.filter(_ belongsTo "John S.")
  .map(_.calculateInterest)
  .filter(_> threshold)
  .foldLeft(0.0)(_ + _)
```

有了闭包、高阶函数等特性的协助，Scala能够定义非常自然的控制抽象，甚至给人感觉就像语言内建的语法一样。即使底层实现是Java对象，也不妨碍我们设计出强力的控制结构，成就简洁明了的DSL。

4. 带给用户的利益

我们在第1章就说过，设计DSL的目的并不是让不懂编程的领域用户用它写代码。所谓设计得当的DSL，重点是API要突显其沟通作用。上一段代码中出现的map、filter、foldLeft等函数式编程的组合子，其实对于领域人员来说并不好理解。但领域人员不难在上述代码片段中看到以下几个要点：

- 过滤出属于John S.的账户；
- 对其计算利息（calculateInterest）；
- 过滤出大于阈值的值；

- 合计所有的利息值。

当你在代码局部集中呈现这些信息要点，领域专家就比较容易理解并验证业务逻辑。如果采取命令式的写法，同样的逻辑很可能散布在较大范围的代码片段里，不懂编程的人很难理清其逻辑。

我们接下来就用Java对象`Account`和代码清单3-5中实现的`AccountDSL`定义一个控制抽象：

```
object AccountDSL {
  def withAccount(trade: Trade)(operation: Account => Unit) = {
    val account = trade.account
    //初始化
      try {
        operation(account)
      } finally {
        //清理
      }
  }
}
```

这段代码用到的两个抽象（`Account`和`Trade`）都是可能经过Scala包装器强化的Java类。现在轮到DSL用户拿这些抽象来做点有用的领域操作了。有刚才的`withAccount`控制抽象，再加上把Scala和Java整合到一起的包装器，用户可以写出下面的DSL代码片段。这段代码照旧比纯Java方案的最好结果表现力更强，更接近于领域用语。

```
withAccount(trade) {
  account => {
    settle(
      trade using
        account.getClient
               .getStandingRules
               .filter(_.account == account)
               .first)
    andThen journalize
  }
}
```

上面一连串的API调用做了些什么，看看图3-7就清楚了。

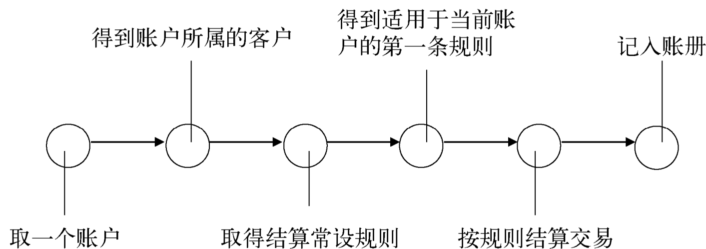


图3-7　前面代码片段的流程图解，分步说明从取得账户到事务结束的全过程

只要几行非常清楚易懂的领域专用代码，`withAccount`就能做这么多事情。如果把这一小段代码拿给领域专家看，他肯定能给你解释其功能。我就拿给老鲍看了。（还记得他吗？这位蹦蹦高证券公司的领域专家从1.4节开始，就一直在给我们帮忙。）你猜怎么着？老鲍看了代码，然

后和我进行了下面这么一番对话。

- 老鲍：你在过滤之后，从结算常设规则里面选其中的第一条，对吧?
- 我：没错!
- 老鲍：但有时候可能有好几条规则适用于同一个账户。
- 我：那你怎么决定应该选哪条规则?
- 老鲍：在这种情况下，每条规则都有对应的优先级。你应该选优先级最高的那一条。
- 我：好!

下回你的经理再说DSL的前期投入太大，你就把刚刚学到的告诉他吧。不见得每一种DSL集成都需要一大笔开支。本小节讨论的包装器方式就是实实在在的例证。这种方式是对当前Java领域模型投资的增值，回报给你的是更灵巧、对领域专家更有用的代码。

只要在Java应用程序中选择Scala作为DSL实现语言，你就可以采用DSL包装器手法。Scala的类型系统有办法把Java对象变得更灵巧，而隐式特性是其秘诀。接下来，我们学习如何利用一些语言特有的集成手段在Java之上实现DSL。这次我们又回到Groovy语言并重温3.2.1节讨论过的DSL示例。

3.2.3 语言特有的集成功能

我们重温3.2.1节集成到核心Java应用程序的那个交易单处理DSL。但这次不用`ScriptEngine`来集成，而是在Java应用程序中动态载入Groovy类的技巧。动态类加载保证Groovy对象即使内嵌于核心Java应用程序，仍然有很好的可操纵性。


元编程、闭包、委托等Groovy知识详见3.7节文献[2]。

1. Java代码与Groovy DSL之间的通信

假设Java交易程序已经用Java 6的`ScriptEngine`集成了交易单处理DSL。一切都运行得很好，直到有一天客户拿来了新的需求：从脚本返回给Java应用程序主体的交易单集合还需要一些额外的处理。具体来说，我们需要计算当前所下全部交易单的总值，还要为客户定制交易单上显示的项目。

在之前的方案里，DSL实现（`ClientOrder.groovy`）和用户脚本（`order.dsl`）被合成一段Groovy脚本，放进`ScriptEngine`的沙盒中执行。Groovy DSL对于Java代码完全不透明；Groovy脚本和Java类分别由不同的类装载器载入，所以脚本内容对于应用程序主体是不可见的。为了满足客户的需求，我们必须找一种办法让Java应用程序接触Groovy类，这就要费一番功夫更换DSL集成方案。

2. 利用Groovy类装载器进行更好的集成

这里，作为DSL实现环节的Groovy类将成为Java应用程序内可重用的抽象，而作为DSL使用环节、由用户编写的交易单处理脚本才由`GroovyClassLoader`加载。代码清单3-6展示的几处修改可令DSL更符合Groovy风格。

代码清单3-6　RunScript.java：利用GroovyClassLoader集成DSL

```
public class RunScript {
    public static void main(String[] args)
            throws CompilationFailedException, IOException,
                InstantiationException, IllegalAccessException {

        final ClientOrder clientOrder =
                new ClientOrder();

        clientOrder.run();                                        ❶ 设置元类

        final Closure dsl =                                       ❷ 加载Groovy类
            (Closure)((Script) new GroovyClassLoader().parseClass(
                new File("order.dsl")).newInstance()).run();
        dsl.setDelegate(clientOrder);                             ❸ 给闭包设置委托
        final Object result = dsl.call();                         ❹ 执行DSL

        List<Order> r = (List<Order>) result;
        int val = 0;
        for(Order x : r) {                                        ❺ 处理结果集合
            val += (Integer)(x.getValue());
        }
        System.out.println(val);
    }
}
```

我们一步步解释这段代码怎样增强DSL集成的Groovy味道。我们分离出ClientOrder.groovy抽象，预编译，使Order类可用于Java应用程序。在上面的Java类中，我们运行ClientOrder的一个实例，以设置元类❶。DSL脚本order.dsl返回一个内含DSL代码的Closure❷。接着，我们设置ClientOrder为Closure的委托，以解析脚本中的符号❸。然后，我们执行DSL脚本，获得一个Order对象的列表❹。最后，我们遍历所有的Order对象，求得交易单总值❺。

DSL脚本一执行完，我们就得到一个Order对象的列表，十分方便后续的业务处理。而按照之前代码清单3-3中的方案，通过Java 6脚本API进行集成，就做不到这一点。客户应该满意这样的结果，而你也学到了一种集成Groovy到Java应用程序的新方法。

3. 最终结果

DSL脚本order.dsl现在变成下面的样子，改为向Java应用程序返回一个Closure。

代码清单3-7　order.dsl：DSL脚本改为返回一个Closure

```
{->
orders = []
ord1 =
newOrder.to.buy(100.shares.of('IBM')) {
  limitPrice   300
  allOrNone    true
  valueAs      {qty, unitPrice -> qty * unitPrice - 500}
}
orders << ord1

ord2 =
```

```
newOrder.to.buy(150.shares.of('GOOG')) {
  limitPrice   300
  allOrNone    true
  valueAs      {qty, unitPrice -> qty * unitPrice - 500}
}
orders << ord2

ord3 =
newOrder.to.buy(200.shares.of('MSOFT')) {
  limitPrice   300
  allOrNone    true
  valueAs      {qty, unitPrice -> qty * unitPrice - 500}
}
orders << ord3

println "Orders ..."
orders.each { println it }
}
```

现在的Groovy交易单处理DSL与第2章中的相比又进步了一些，而且它与Java集成的效果也比3.2.1节的`ScriptEngine`方案更出色。

介绍完语言特有的集成功能，下面介绍一种基于框架的内部DSL集成方案。Spring提供了这样的平台，我们来看看它的用法。

3.2.4 基于Spring的集成

表3-2总结的集成手法就剩下最后一种未介绍了。本节要介绍的集成方案通过框架来实现，论抽象层次，它比前面那些语言层面的集成方案更高。要是Java应用程序里面的业务规则可以动态修改，而且不需要重新启动程序，那该多好；你会不会常常这样想？

1. Spring的动态语言支持

从2.0版开始，Spring即支持用Ruby、Groovy等表现力强的动态语言所实现的bean。（欲详细了解Spring，请访问http://www.springframework.org。）这类bean有所谓的“可刷新”性质，即当其底层实现发生变化的时候，可以动态地重新装载它们。来看一个金融中介领域的例子。假设有个`TradingService`实现，为了计算附息债券的应付利息，需要查找一些计算规则。

```
public class TradingServiceImpl implements TradingService {
    private AccruedInterestCalculationRule accIntRule;    <-- ❶ 由Spring注入的计算规则

    @Override
    public void doTrade(Trade trade) {
        // 具体实现
    }
}
```

在上面的代码片段中，应付利息的计算规则经由Spring DI在运行时注入❶。利用Spring对动态语言的支持，我们可以选择JRuby、Groovy、Jython等表现力好的语言来实现这些规则。此处的场景正好适合一种小巧、内涵丰富的DSL发挥作用。这样有两个好处：

- 因为语言本身内涵丰富，代码的表达更到位；

- 当bean的底层实现发生变化时，其运行时实例可以自动重新加载。

就当前的例子而言，我们用个Java接口来定义计算规则的契约：

```
public interface AccruedInterestCalculationRule {
    BigDecimal calculate(Trade trade);
}
```

然后规则的具体实现可以用Ruby DSL来写：

```
require 'java'

class RubyAccruedInterestCalculationRule {
  def calculate(trade)
    //具体实现
  end
end

RubyAccruedInterestCalculationRule.new
```

现在就剩最后一件事情了。

2. 接通实现

用下面的Spring XML配置片段就可以把整个实现连接起来。现在，当Java程序需要一个`AccruedInterestCalculationRule`实例的时候，就会得到由Ruby DSL编写而成的实例。

```
<lang:jruby
  id="accIntCalcRule"
  refresh-check-delay="5000"
  script-interfaces=
    "org.springframework.scripting.AccruedInterestCalculationRule "
  script-source="classpath:RubyAccruedInterestCalculationRule.rb">
</lang:jruby>
```

恭喜你，你成功利用Spring在Java应用程序中集成了Ruby DSL。这种非侵入式的DSL集成模型使DSL组件与使用它的上下文解耦。如果你的应用程序正好将Spring用作DI框架，不妨考虑用这种集成模式解决业务规则DSL动态重新加载问题。

针对内部DSL的同质集成模式至此介绍完毕，我们接着学习外部DSL的集成模式。外部DSL形态各异，其语言设施也可能是专门设计的。下一节，我们会回顾2.3.2节讨论过的所有外部DSL实现模式，看看不同的实现方案会在核心应用程序上留下怎样的集成入口。注意，外部DSL都是特别为了某个应用程序而定制的；我们对于外部DSL集成模式的讨论受限于几种常见的使用手法。

3.3　外部 DSL 集成模式

怎样在应用程序中集成XML？你会立即回答：“使用XML解析器!”没错！因为XML不同于应用程序的宿主语言，所以需要单独的解析和处理机制。XML应用非常广泛，相应地工具非常多，比如XPath、XQuery等，而且几乎随便一个企业解决方案都会带上各式各样XML解析器。在应用程序中集成XML简直轻而易举。然而很遗憾，我们为应用程序设计的外部DSL没有这样的“全套行头”。集成我们的外部DSL到应用程序更多地依赖于特殊情况下的特殊举措，难以推广成通用的模式。

基于上一段中的说法，你恐怕觉得集成外部DSL会是软件开发中的一场噩梦。是否如此则取决于DSL的复杂度和实现技术。如果外部DSL的解析器是采用ANTLR、YACC等标准工具来开发的，那么集成起来还是很简单的。如果重读一遍2.3.2节，你会发现那里描述的每一种外部DSL实现模式，都为其设计产物留下了显而易见的集成入口。

那我们就再细数一遍2.3.2节中的外部DSL模式，试试找出它们的集成入口。表3-3对于如何集成外部DSL到应用程序作了总结。

表3-3　外部DSL的集成入口

外部DSL模式	集成入口
上下文驱动的字符串操控	字符串经过分词处理被转化为宿主语言代码，这一过程中用到了正则表达式匹配和动态代码解释等技术。转化得来的代码片段就是与应用程序集成的入口
XML转换成可使用的资源	XML解析器就是最自然的集成入口。经过解析，XML被转化成宿主语言中的数据结构，可被应用程序直接使用
非文本表示	非文本表示被转化成AST。以AST为基础，我们可以生成多种形式的具体语法树。只要根据应用本身使用的宿主语言生成一棵该语言的具体语法树，我们就有了集成入口
DSL中内嵌异质代码	DSL处理引擎按内嵌代码所属语言把DSL转化成该语言的适当数据结构，同时将各段内嵌代码作为回调函数插入其中。结果得到内嵌代码中的一组数据结构，由于语言相同，可以直接被核心应用程序使用
基于解析器组合子的DSL设计	在Scala等语言中，解析器组合子被实现为库。以宿主语言写成的组合子就是解析外部DSL的规则。利用一些嵌入的宿主语言代码，规则一边解析，一边填充语义模型的数据结构。当规则约减到AST的最高节点，我们就得到了完整的DSL语义模型

为何外部DSL集成模式讲解得不如内部DSL集成模式那么详细？内部DSL集成只需要倚靠宿主语言，而外部DSL根据其领域常需要种类不确定的、数量又比较多的全套设施。比起用宿主语言设计一套API，语言处理设施的设计没有一定之规。因此，若脱离了具体的环境和要求，我们很难一般性地讨论外部DSL集成模式。第7章和第8章将以具体示例来详细介绍这方面的技术。



第7章讨论如何用ANTLR设计DSL，ANTLR是常用的解析器生成器。我们还介绍了利用DSL工作台的一些外部DSL生成工具。另外，针对采用ANTLR和DSL工作台两种方式下产生的外部DSL，我们还介绍了如何将DSL与核心应用程序集成。

第8章详细介绍了如何利用Scala解析器组合子设计外部DSL。

我们已经介绍完毕内部DSL和外部DSL的所有集成模式，这些模式足以应对工作中的大部分情况。全篇讨论都假定核心应用程序是以Java开发的，而准备集成的DSL则是以表现力更强的语言写成的。这个假设符合现实中最常见的情况，所以请务必充分理解本章讨论的这些集成问题。

本章一开篇就在图3-1中展示了DSL驱动应用程序开发中头等重要的问题：集成DSL到核心应用程序。但不时有些开发者忘了提前规划这个问题，直到开发后期才开始考虑。我们即将讨论的下一个问题，也时常在起始阶段被开发者忽略：决定错误及异常处理的策略。这是一件应该优先去做的事情，尤其是DSL的用户基数比较大的时候。

3.4　处理错误和异常

给用户看到友好的错误报告，其重要性绝不低于提高DSL语法的表现力。因为DSL是一种应用范围受限的语言，错误消息也应该适应其应用范围，用领域语言去表达。DSL环境内的错误和异常报告要有章可循，不能误导用户和造成困惑。报告中要清楚说明系统所处的确切状况。这种设计思路叫做领域驱动的异常报告，详见3.4.1节。我们还会讲到DSL用户可能面对的两种主要类型的错误状态。以上几点要求共同构成了错误和异常处理策略的三大支柱，是DSL设计者不可忽略的参考视角，如图3-8所示。

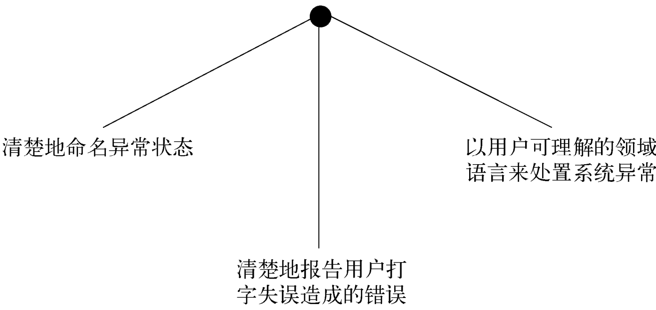


图3-8　DSL错误和异常状态处理策略的三大支柱

根据DSL的内部、外部之分，以及实现语言的区别，错误状态的呈现方式也有所不同。表3-4总结了有关错误、异常的待解问题，还有你身为DSL设计者的责任。

表3-4　关于DSL错误和异常，你应该知道的一些事

待解问题	DSL设计者的解决之道
你需要清楚声明DSL内的异常状态	异常状态也是领域抽象。应当一直使用领域语言来表述过程中可能发生的任何异常。详见3.4.1节
当用户输错了方法名、对象名或者其他语言成分，你需要处理因此产生的错误	具体策略取决于所用的实现语言。详见3.4.2节
当系统进入无效的业务状态，你需要处理因此出现的异常。例如，当与银行的通信中断时，若有人试图转账一笔资金，会发生什么事呢	在报告此类异常的时候，请务必以用户可以理解的语言提供所有相关详情。详见3.4.3节

下面我们就详细探讨这几个问题。

3.4.1　给异常命名

给DSL里的异常状态命名的时候，我们应该采用领域用户的用语描述那种情况。异常不一定是设施出了毛病，可能只是业务用例中的一条分支。命名的重点是通过领域语汇将情况呈现出来。下面的例子来自一个对交易双方账户进行结算的系统：

```
val fromBalance = fromAccount.getSecurityBalance
if (fromBalance <= tradeQuantity)
  throw new SettlementFailedException(
    " Insufficient security balance in " +
    "account " + counterpartyAccount.getName +
    " for settlement completion")
settle(..)
```

系统遇到了异常状况；当卖家的证券余额不足的时候，结算将失败。事件的状态已经通过异常名SettlementFailedException表达出来，其措辞也符合现实中结算系统的一般用语。当用户看到这个异常，他立即就会明白当前的情况。而且，他还会在异常附带的消息里看到详细的失败原因。

3.4.2 处理输入错误

不管DSL语言多么自然，用户还是会写错，这是人类本性。如果所用编程语言是像Scala和Java这样的静态类型语言，编译器会在错误发生时立即给你提醒。只要你违反了语言类型系统的规则，编译器就会像警察一样来警示你，如图3-9所示。



图3-9 编译器就像警察一样发挥警示作用

如果用户对实现DSL的宿主语言有足够的了解，编译器报告的错误消息将很有帮助。现代IDE都带有代码助手和自动补全功能，有利于防范输入错误。可是，如果没有这些帮助又该怎么办呢？

1. 当类型系统不可用时

像Ruby和Groovy那样的动态类型语言没有编译器帮忙找出类型错误。在这些语言里，类型错误大多要经过语言的方法分发流水线处理之后作为运行时错误呈现出来。即使没有编译时的错误检查，也不妨碍设计得当的DSL发挥动态语言的优势来达到目的，比如利用methodMissing特性来设置友好的错误处理程序，向DSL用户传达纠正错误所需的信息。

对于使用动态语言来设计DSL而言，采用methodMissing是非常有用的技巧。下面的Ruby示例就为方便用户理解运行时异常补充了足够的上下文信息：

```
class Trade
  //..
  def method_missing(method, *args, &block)
```

```
    raise NoMethodError, <<ERRORINFO
method: #{method}
args: #{args.inspect}
on: #{self.to_yaml}
ERRORINFO
  end
  //..
end
```

如果用户输入了Trade对象中不存在的方法名，Ruby将默认抛出一个NoMethodError。而上面的代码片段实现了method_missing来充当定制的错误处理程序，可以向用户提供更多上下文信息。（至于在Groovy中利用methodMissing合成新方法的例子，请翻阅2.2.2节。）

2. 语法解析器的功用

对于外部DSL，解析器在解析输入脚本的过程中需要确保报告输入字符串发生错误的准确行号和位置。错误报告的友好程度非常依赖于生成DSL语法解析器所采用的技术。在错误报告方面，ANTLR生成的自顶向下型解析器比YACC的自底向上型解析器支持性更好。在第7章讨论通过解析器生成器设计外部DSL的时候，我们会详细解说这部分内容。

好了，对于注定要出现的用户输入错误，你已经知道该怎样对付了。那么，如果业务状态出了错误，该怎么办呢？

3.4.3 处理异常的业务状态

DSL应该有能力精确报告异常状态，且按照3.4.1节所说的“领域驱动的异常报告”方式进行。比报告异常更重要的是处理异常。DSL运行期间可能抛出的所有领域异常，都应该有相应的处理程序，包括实施各种清理动作、释放资源和回滚事务。

怎样处理和报告异常才算合适，这也要看你选择什么样的策略来集成DSL脚本。像3.2.1节中那种依靠ScriptEngine的集成策略，一般在报告异常方面表现不好。下面的例子来自我们之前讨论在Java应用程序中内嵌Groovy脚本的部分，我们看看它是怎么报告异常的：

```
ScriptEngineManager factory = new ScriptEngineManager();
ScriptEngine engine = factory.getEngineByName("groovy");

try {
  List<?> orders = (List<?>)
  engine.eval(new InputStreamReader(
    new BufferedInputStream(
      new SequenceInputStream(
        new FileInputStream("ClientOrder.groovy"),
        new FileInputStream("order.dsl")))));
} catch (javax.script.ScriptException screx) {        ❶ 处理异常
  // 具体的处理
}
```

示例并没有在Groovy代码内显式处理异常，我们只是假设脚本的调用者会去处理❶。javax.script.ScriptException类带有getFileName()、getLineNumber()等方法，有助于找到发生异常的确切位置。凡是源自DSL内部的异常都必须谨慎处理，而且你需要向用户提供

充分的上下文信息——这是重要的处理原则。然而，当DSL代码运行在`ScriptEngine`的沙盒里面时，处理异常的时候所需的上下文信息不一定直观。这个缺点再次说明，集成DSL应该优先选择语言专门提供的方式，只在不得已时选择Java脚本引擎方案。

DSL的设计宗旨是领域内的可读性和表现力，因而我们必须时刻考虑到领域用户。与此同时，我们也要留心DSL对应用设计的性能有何影响。应该如何折中呢？

3.5　管理性能表现

性能是重要的评判标准，但不管你怎么想，我认为它并不是最重要的评判标准。性能低下的应用程序其性能可以通过横向或纵向增加资源来提高；但是，如果实现了一个完全不关心沟通和维护性的混乱系统，你将注定受困于它。

话说回来，设计应用程序的时候你还是很有必要考虑性能因素的。实事求是地说，良好的DSL设计不见得一定拖累应用程序的性能表现。有些动态语言，如Groovy和Ruby，确实比Java慢一点。但应用程序开发者和架构师需要在速度和其他特质之间取舍，考虑代码的可维护性、表现力、对未来变化的适应性等，并在它们与速度之间进行权衡。

应用程序并非每一部分都需要快如闪电，有些部分的维护性比速度更重要。例如，应用程序的配置参数一般只需要处理一次，这可能是在应用程序启动的时候进行。所以，与将配置写入代码来加速应用程序启动相比，我们不如将一部分配置参数外部化，以更易读的形式呈现给用户。改善表现力的好处远远大于应用程序速度损失可能导致问题的坏处。

目前着力于改善JVM上动态语言执行性能的自发行动十分活跃，因此我们更应该选择这些语言来设计DSL。如果现在就注重提高代码的表现力，等到Groovy和Ruby的语言运行时在JVM上的性能提高了，我们的代码也能自动享受到性能改善的益处。

大家完全清楚Groovy和Ruby代码比相同功能的Java代码要慢一些，要是没有好处，谁会用它们来设计DSL呢？这些语言好维护、易读，而且能很好地适应变化，而当宿主语言本身具备充足的表达能力时，DSL的成长历程将顺利得多。我们并非贬低性能的重要性，只是说这些因素和单纯的执行速度同等重要。你设计的领域语言，其演变道路和生命线由所有这些因素共同决定。

像Scala那样的静态类型语言性能几乎等同于纯Java。3.2.2节介绍的DSL包装器集成模型，其性能基本和纯Java应用程序没有区别。

脚本引擎因为运行在沙盒环境下，多少会慢一些，不过反正你也不会用脚本来执行强调性能的任务。脚本式的DSL主要用于处理轻量级领域逻辑，使用者也以最终用户和领域专家为主。内嵌式DSL（内部DSL）主要实现成宿主语言的库，所以并不会拖累性能。外部DSL没有依靠和束缚，可以自由实现其语言机制。在大多数现实的应用程序中，外部DSL不需要设计得像完整的高级语言那么复杂，而且有解析器生成器（YACC、ANTLR）和（Scala、Haskell语言中的）解析器组合子等工具帮忙，如果善加利用，并不难构建出必要的语言设施。

最后，请记住一条性能调优的黄金法则：多点基准测试，慢点优化性能。

3.6 小结

本章我们从所有的角度出发讨论了DSL驱动的应用程序开发，介绍了怎样选择恰当的策略来集成DSL和核心应用程序。通过其中介绍的各种集成模式，相信你已经了解什么时候选用包装器模式，什么时候选用脚本引擎模式。在相当程度上，你选择的实现语言决定了最恰当的集成策略。我们还谈到怎样处理错误和异常，如何以符合领域习惯的用语向用户报告错误和异常。最后，我们讨论了DSL代码的可维护性和性能表现之间的取舍。

要点与最佳实践

- **DSL从不单独存在**。它们必定与核心应用程序集成在一起。如果你打算设计DSL，请从第一天起就牢记这条黄金法则。
- **设计内部DSL的时候，DSL的实现语言应该选择与应用程序的核心语言集成效果最佳的那一种**。
- **外部DSL通常需要额外的设施，如解析器生成器**。在计划阶段你就应该预作打算，确保团队拥有相应的开发资源。
- **集成DSL与核心应用程序时，你应该遵从经过考验的最佳实践**。

看完这一章，本书的入门部分就要结束了。后面的章节，我们将深入介绍DSL实现的各方面内容。我们将探讨多种JVM语言，用每一种语言设计、实现各种DSL代码片段，并讲评每一种方式的优缺点。后面有一段精彩纷呈的旅途在等待着你。准备好，保持冷静，我们要出发了。

3.7 参考文献

[1] Odersky, Martin, Lex Spoon, and Bill Venners. 2008. *Programming in Scala*. Artima.

[2] König, Dierk, Paul King, Guillaume Laforge, and Jon Skeet, 2009. *Groovy in Action*, Second Edition. Manning Early Access Program Edition. Manning Publications.

[3] Ghosh, Debasish. Why I like Scala's Lexically Scoped Open Classes. *Ruminations of a Programmer*. http://debasishg.blogspot.com/2008/02/why-i-like-scalas-lexically-scoped-open.html.

Part 2 第二部分

实现 DSL

从表面来看，DSL 的语法和领域用户的日常生活中的用语一致。本书第一部分着重强调让软件“说”领域语言的重要性。而当你做到了第一部分的要求之后，还有 DSL 语法背后的语义模型等着你去培育，按照抽象的设计原则去塑造它。除非语义模型易于扩展、易于适应、易于组合，否则建立在语义模型之上的语法很难有出色的表现力。

第二部分（第 4 章 ~ 第 8 章）讨论能塑造出优秀语义模型的所有惯用法和最佳实践。

设计 DSL 的时候，你要按照编程所需的抽象层次，找到最适合表达该层次抽象的语言。这一部分将会使用 Groovy、Ruby、Scala 和 Clojure 语言来实现 DSL。这些语言各有长处和短处，也各有其特色功能可用于 DSL 组件建模。若追求基于 DSL 的开发方式，你就有必要了解这些语言提供的惯用法，还有它们与主应用架构结合的方式。

该部分还涵盖了基于 ANTLR 和（来自 Eclipse 的）Xtext 等现代框架的外部 DSL 开发。ANTLR 是一种语法分析器生成器，可以帮助 DSL 编写定制的语法分析器。Xtext 是完整的外部 DSL 开发及管理环境。

该部分的压轴主题是分析器组合子，这是一种用于外部 DSL 开发的优美的函数式抽象。

第 4 章

内部DSL实现模式

本章内容

- 内嵌式DSL的元编程模式
- 内嵌式DSL的类型化抽象模式
- 生成式DSL的运行时元编程模式
- 生成式DSL的编译时元编程模式

本书第一部分已经领你步入了DSL驱动的开发范式。我们一起见识了DSL的风采，也见识了它在现实应用中可能出现的问题。从本章开始，我们开始探讨DSL实现层面的内容。

每一位架构师都有个“工具箱”，里面放着自己用来雕琢优美软件制品的趁手工具。经过这一章的学习，你会拥有自己的“工具箱”存放一套用于实现DSL的架构模式。图4-1简单列举了本章打算涉及的话题。

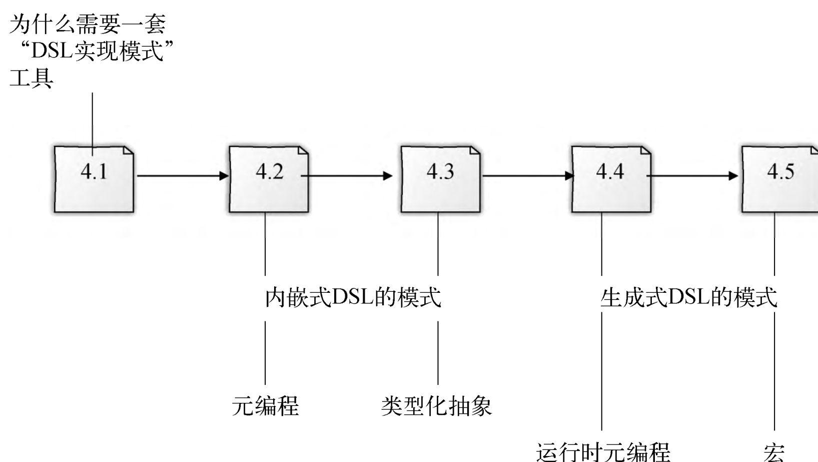


图4-1　本章路线图

DSL设计者绝不可忽视DSL实现中的惯用法和最佳实践。因此，我们首先介绍一系列可以用于现实开发工作的模式。内部DSL一般都内嵌于某种宿主语言，而充当宿主的语言往往支持某种

元对象协议（meta-object protocol），可以用来在DSL上实现一些动态行为。内部DSL的实现语言大多是动态类型语言，如Ruby和Groovy，我们会在4.2节讲解几种利用它们的元编程能力的几种模式。静态类型语言的抽象能力可用于将DSL建模成宿主语言的内嵌类型，4.3节以Scala作为实现语言讲解这类模式。4.4节和4.5节讨论不同语言的代码生成能力，它们可用于实现简练的内部DSL。这样产生的DSL称为生成式DSL，因为它们简洁的表面语法所代表的领域行为，是在编译时或运行时通过生成宿主语言的代码来实现的。学习完本章，你定会感觉胸有成竹，因为你的“工具箱”里将塞满各种实用的问题域建模技巧、模式和最佳实践。

4.1　充实 DSL“工具箱”

工匠大师总是随身带着塞得满满的工具箱。箱里最开始的几件工具是他们从师傅那里继承来的，然后是其靠自己在一年一年的历练中逐渐充实进去的。这本书会交给你一些适合放进“箱子”里的DSL工具，尤其是实现内部DSL的工具。

我们接着2.3.1节讨论的内部DSL一般模式往下说。它们都是一些实现模式，适用于不同的DSL设计场景。2.3.1节用了不少代码片段来演示这些模式在常用语言中的呈现形式。本章将延续前面的讨论，举出金融中介系统这个问题域的例子，尝试将问题域的场景联系到它们在解答域的对应实现。阅读的时候，请留心收集可用于充盈你“工具箱”的工具吧。有时候，我会演示同一模式的不同实现手法，一般不同手法所用的实现语言也不同，重点说明每种手法涉及的权衡取舍。

展开讨论之前，我们先看看图4-2。图4-2中的模式还是我们在第2章看到的那些，但这次标注了每种模式对应的实现制品形态，也就是本章要讨论的内容。其中有一处标注需要说明，请看内嵌式DSL类别下的一个模式——“反射式元编程”方框。在本章的讨论中，我们会用这种模式实现Ruby和Groovy的隐式上下文（implicit context），还有Ruby的动态装饰器（dynamic decorator）。两种实现制品都有利于DSL服务于其用户：无需增加任何不必要的复杂性，用户就能清晰地表达意图。

如图4-2所示，内部DSL分为两类。

- **内嵌式DSL**　DSL寄身于宿主语言，这意味着所有的DSL代码都是程序员直接写出来的。
- **生成式DSL**　部分DSL代码（大多为重复性的内容）由编译时或运行时语言机制生成。

在现实的应用中，模式不会单独出现。一般，多种模式协同作用，构成配套方案，合力解决用例场景。一种模式的作用产物被另一种模式所承接，整个系统环环相扣，就像“模式语言”在进行一场融洽对话。后面的讲解风格会按照DSL设计者的工作思路来安排。也就是说，我不打算像字典那样工整地分别介绍每种模式，而是先举出金融中介系统领域的DSL代码片段示例，然后剖析其中的模式结构。这样你不但能看到每种模式在现实用例中的呈现方式，还能体会不同模式结构如何协同形成更大的整体。

我们首先看看适用于实现内嵌式DSL的模式结构。

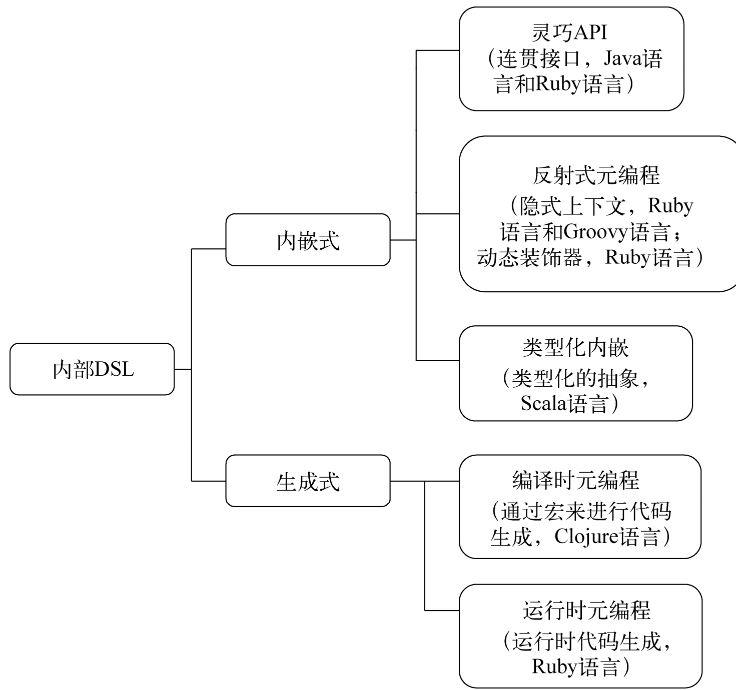


图4-2　内部DSL实现模式及其制品示例。本章将讨论这些制品，并按图中所列语言提供相应的示例实现

4.2　内嵌式 DSL：元编程模式

元编程就是“编写写程序的程序”。书本上像这样的元编程定义，很容易使人误以为元编程不过是给代码生成换了个花哨的名字。从实践角度来说，元编程，是在语言环境的编译时或运行时设施上，用设施所提供的元对象来撰写程序。看过2.5节的深入讨论，相信你能理解这个定义，我们就不再赘述了。

本节我们将观察金融中介系统领域的几个例子，它们都可以用元编程手段来建模。对于一部分例子，我会首先用一种不具备元编程能力的语言来实现；然后，当我们换一种语言，发挥其元编程能力在更高层次的抽象上编程时，你会看到前后两种实现简洁度的变化。

代码提示　后面的内容中含有大量代码片段，我会插入补充内容说明一些预备知识，解释必要的语言特性，以便读者理解实现中的细微之处。这些补充内容只是简单说明一下稍后代码清单中就要用到的语言特性，欲了解特定语言的更多相关信息，请参阅本书附录中相应语言的速查表。

本节接下来给出了3种模式风格，这些风格落实之后，就成为图4-2列出的那些元编程模式的具体实现。我们将从一个用例场景开始，从用户的角度去观察其中的DSL，并且解析DSL来了解其实现结构。一个用例不见得只应用了一种模式，实际上以下每种模式风格之下，都为了履行解答域的职责，准备了好几种具体的模式实现。

4.2.1 隐式上下文和灵巧API

我们先对前面的章节来个简短回顾。客户在证券交易商那里开户，证券交易商代客户交易他们持有的股票并保证安全。关于客户账户的更多信息，请翻阅3.2.2节的补充内容“金融中介系统：客户账户”。

下面我们准备设计一段交易商开设客户账户的DSL。你可以自行评判元编程技巧对API表现力的改善效果，即使这些元编程技巧作用于用户看不到的内部实现。

1. 评判DSL的表现力

来看下面这段DSL脚本。它的功能是创建新的客户账户，然后向证券中介企业注册该账户。



Ruby知识点

- **Ruby怎样定义类和对象**。Ruby是一种面向对象（OO）语言，它定义类的方式和其他OO语言差不多。不过，Ruby有其独特的对象模式，允许用户在运行时通过元编程手段修改、检查、扩展对象。
- **Ruby用“块”（block）来实现闭包**。闭包指的是一个函数和它的执行环境。Ruby通过它的“块”语法实现闭包。
- **Ruby元编程基础知识**。Ruby的对象模型包含许多可以用于反射式和生成式元编程的元件材料，例如类、对象、实例、方法、类方法、单例（singleton）方法等。Ruby元编程机制允许你在运行时探查其对象模型，也允许动态地改变对象行为或生成代码。

代码清单4-1 创建客户账户的DSL

```
Account.create do                                    ❶ 创建账户

  number       "CL-BXT-23765"
  holders      "John Doe", "Phil McCay"
  address      "San Francisco"
  type         "client"
  email        "client@example.com"

end.save.and_then do |a|                             ❷ 保存账户

  Registry.register(a)
  Mailer.new
        .to(a.email_address)
        .cc(a.email_address)
        .Subject("创建新账户")
        .Body("客户账户 #{a.no} 已创建")              ❸ 开户后发送邮件
        .send
end
```

上面的代码展示了用户使用DSL的情况。留意观察它是怎样隐藏实现细节，同时又将账户创建过程向DSL使用者表达清楚的。这段代码创建账户之余还做了其他一些事情。你能够在不知道内部实现的前提下，仅从代码清单4-1看出是哪些事情吗？如果你全都能看出来，那么这就是一段漂亮的DSL实现。

你很容易识别这段DSL代码的全部举动。它首先创建账户❶并保存（可能是保存到数据库）❷，然后是登记账户并发送邮件给账户所有人❸等动作。

从表现力上看，这段DSL给用户提供了一套符合直观感觉的API，效果很好。领域专家拿到这段DSL脚本肯定也能看出里面的动作序列，因为脚本中很好地运用了领域语汇。接下来，我们开始深入了解内部实现。

2. 定义隐式上下文

`Account`抽象内实现了创建实例时需要调用的众多方法。代码清单4-2中完全是一般Ruby语言定义类实例方法的常规写法。

代码清单4-2　`Account`抽象在其实现中运用了领域语汇

```
class Account
  attr_reader :no, :names, :addr, :type, :email_address

  def number(number)
    @no = number
  end

  def holders(*names)
    @names = names
  end

  def address(addr)
    @addr = addr
  end

  def type(t)
    @type = t
  end

  def email(e)
    @email_address = e
  end

  def to_s()
    "No: " + @no.to_s +
    " / Names: (" + @names.join(',').to_s +
    ") / Address: " + @addr.to_s
  end
end
```

这些显而易见的方法定义不是我们关心的内容，我们要关注这些常规语句表现出来的微妙方面，发现前面所说的那些实现模式。

拿到一段代码，你首先要确定它的执行上下文。这个上下文可以是刚才定义的一个对象，也可以是你特地配置或者隐式声明的一个执行环境。有的语言要求对于每一处方法调用都明确将方

法和对应的上下文联系在一起。请看下面的Java代码：

```
Account acc = new Account(number);
acc.addHolder(hName1);
acc.addHolder(hName2);
acc.addAddress(addr);
//..
```

所有针对Account对象的方法调用都必须在前面带上它的调用者，以此显式地传递上下文对象。显式表达上下文会产生繁冗的代码，不利于我们创作简洁易读的DSL。如果一种语言允许隐式声明上下文，肯定不是坏事。隐式上下文有利于语法简明，API紧凑。代码清单4-1中的number、holders、type、email等方法调用都发生于一个隐式上下文中，也就是正在创建的Account实例之内。

那么怎样建立隐式上下文呢？请看下面从Account类中截取的相关Ruby代码片段，它使用一点巧妙的元编程手法达到了目的：

```
class Account
  attr_reader :no, :names, :addr, :type, :email_address

  ## 省略部分同代码清单4-1

  def self.create(&block)                    ❶ create接收一个块
    account = Account.new
    account.instance_eval(&block)            ❷ 在Account的上下文内执行传入的块
    account
  end
end
```

请看位置❷，instance_eval是一种Ruby元编程语法结构，它会在其调用者的上下文内执行传递给它的Ruby块，因为我们是在Account对象上调用的，所以块就在Account对象的上下文内执行。结果对于那些通过块进行传递的方法❶来说，就好像每次调用的时候都隐式地接受了刚刚构造完毕的account对象。这是一个反射式元编程的例子。Ruby执行环境在运行时通过反射确定执行的上下文。

相同的手法也适用于Groovy，Groovy语言同样具备很强的元编程能力。

我们用Groovy语言改写上述Ruby代码，结果见代码清单4-3。



Groovy知识点

- 如何在Groovy中创建闭包并为方法分发准备上下文。

代码清单4-3　为方法分发准备隐式上下文的Groovy代码

```
class Account {

  // 方法定义

  static create(closure) {
    def account = new Account()
    account.with closure
    account
```

```
  }
}

Account.create {
  number      'CL-BXT-23765'
  holders     'John Doe', 'Phil McCay'
  address     'San Francisco'
  type        'client'
  email       'client@example.com'
}
```

前后两种实现不太一样，但得到的API表现力差不多。

3. 利用灵巧API改善表现力

易读是改善DSL表现力的必然结果。*连贯接口*是提高可读性、实现灵巧API的途径之一。通过方法串联，一个方法的输出很自然地成为另一个方法的输入。这种手法使连串的API调用表达起来更自然，也比较接近问题域内真实的动作序列。同时，因为调用的时候摆脱了一些死板代码，API显得“灵巧”。

如果回顾代码清单4-1中发送邮件之前的连串方法调用❸，你会看到那里的API调用跟你平常使用邮件客户端软件的动作序列是一样的。

你在设计DSL的时候应该注意语句是否流畅。代码清单4-4中的代码片段实现了代码清单4-1中使用的`Mailer`类。

代码清单4-4　实现了连贯接口的`Mailer`类

```
class Mailer
  attr_reader :mail_to, :mail_cc, :mail_subject, :mail_body

  def to(*to_recipients)
    @mail_to = to_recipients
    self
  end

  def cc(*cc_recipients)
    @mail_cc = cc_recipients
    self
  end

  def subject(subj)
    @mail_subject = subj
    self
  end

  def body(b)
    @mail_body = b
    self
  end

  def send
    # 实际的发送操作
    puts "发送邮件到(#{@mail_to.join(",")})"
  end
end
```



❶ 返回自身以利串联

Mailer实例被返回给调用者❶充当下一个调用的上下文。send方法是方法链条的最后一环，它结束整个动作序列，把邮件最终发送出去。

代码清单4-1向我们展示了创建账户的DSL，开户的3个步骤在代码中已经比较明显。不过，图4-3把步骤呈现得更清楚，它们运用各种模式塑造了DSL的最终形态。

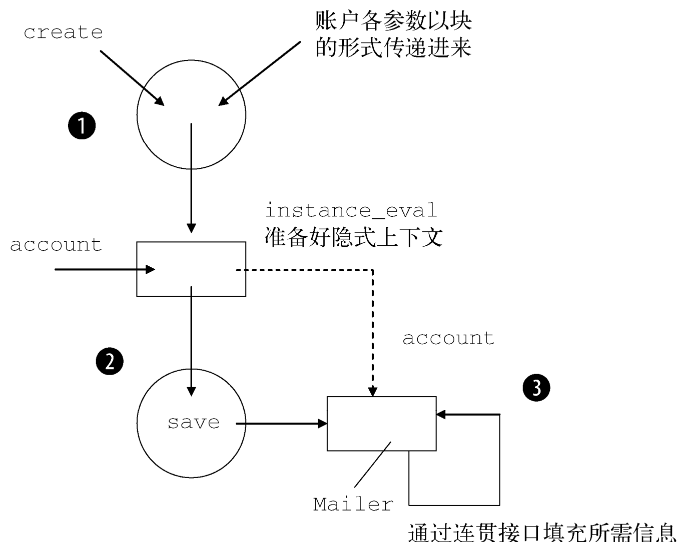


图4-3　模式的应用步骤：❶在通过instance_eval设立的隐式上下文里创建账户。❷保存账户。❸通过连贯接口配置好Mailer，账户通过一个块来传递

模式的应用分为如下3个步骤：创建一个账户实例，将之保存到数据库，执行其余后续操作。在这里，第三步被设置成一个闭包（或Ruby块）。之前创建的account实例被作为输入传递给这个闭包，用于执行其他的操作。account实例在操作完成后仍然保持不变；注意，第三步操作属于有“副作用”（side-effecting）的操作。

管理程序的副作用是一个微妙的程序设计问题。我们需要将副作用隔离以保持抽象的纯粹性。无副作用的抽象可以为你减少许多烦恼。设计DSL的时候，你应该尽量明确地隔离有副作用的操作。代码清单4-1就把所有涉及副作用的代码解耦出来，放到了一个闭包中。隔离副作用的设计思路不仅适用于内部DSL设计，你设计任何抽象都应该牢记这个原则。

本节我们讨论了基于DSL的设计范式下两种被普遍使用的实现模式。本节的要点见下面的补充内容。

本节要点

通过方法串联手法**实现带连贯接口的灵巧API**（参见代码清单4-4中的Mailer类）。

隐式上下文可降低DSL的烦琐程度，形成比较紧凑的API，帮助提高表现力（参见Ruby

代码片段中的`create`类方法，或者代码清单4-3中Groovy代码片段里的`create`静态方法）。

把副作用跟纯粹的抽象隔离开（参见代码清单4-1中用于开户和发送通知邮件的Ruby块）。

接下来，我们继续介绍别的实现结构，它们通过反射式元编程在DSL中实现动态行为。

4.2.2　利用动态装饰器的反射式元编程

在4.2.1节，我们了解到元编程技巧可以用来改善DSL的表现力和简洁度。本节将介绍运行时的另一种元编程手法：通过动态操控类对象，对其他对象进行装饰。

装饰器（Decorator）模式用于在运行时动态地增加对象的功能。（装饰器模式用在抽象之间，可以增加它们的组合能力，附录A对此有所讨论。）本节我们偏重于实现的角度，看看怎样发挥元编程的威力，做出更动态的装饰器。

1. Java中的装饰器

我们还是从`Trade`抽象入手，这个领域实体是对交易过程中一些最基本相关要素的建模。作为例子，我们打算在`Trade`对象的基础上设计一些影响其交易净值的配套装饰器。关于如何计算交易的现金价值，请看补充内容“金融中介系统：交易的现金价值”。



金融中介系统：交易的现金价值

每一笔交易都有其现金价值，也就是接受证券的交易方需要向交出证券的交易方支付的金额。这个最终的支付金额称为NSV（Net Settlement Value，净结算价值）。NSV主要由两部分构成：证券总值和税费。

证券总值取决于所交易证券的单价、种类，还有一些附加成分，如债券的孳息价格。税费额需要计入各种税、手续费、交易征费、佣金以及交易过程中产生的利息等。

证券总值的计算与证券的类型有关（比如有股权和固定收益之分），但大体上是所交易证券的单价和数量的一个函数。

另外的税费部分，根据交易发生的所在国、所在交易所、交易的证券，各有不同规定。例如，在中国香港，印花税被定为0.125%，买入和卖出股权证券要交0.007%的交易征费。

请看代码清单4-5中的Java代码。

代码清单4-5　`Trade`抽象和它的Java装饰器

```
public class Trade {                          <-❶ Trade抽象
    public float value() {                    <-
        // ..                                  ❷ 计算并返回交易价值
    }
}
```

```
public class TaxFeeDecorator extends Trade {
    private Trade trade;

    public TaxFeeDecorator(Trade trade) {
        this.trade = trade;
    }
    @Override
    public float value() {
        return trade.value() + //..;
    }
}

public class CommissionDecorator extends Trade {
    private Trade trade;

    public CommissionDecorator(Trade trade) {
        this.trade = trade;
    }
    @Override
    public float value() {
        return trade.value() + //..;
    }
}
```

❸ 装饰器

❹ 税费计算的具体细节

❺ 佣金计算的具体细节

4

代码清单4-5除了实现Trade抽象的契约❶，还有两个配套的装饰器❸，装饰器与Trade搭配使用可影响交易的成交净值❷❹❺。这些装饰器的用法如下：

```
Trade t =
    new CommissionDecorator(
        new TaxFeeDecorator(new Trade()));
System.out.println(t.value());
```

你完全可以在不触动基本的Trade抽象前提下，在其上继续增加其他装饰器。最后计算出来的交易净值等于施加于Trade对象的所有装饰器的合并作用结果。

代码清单4-5就是用Java实现的，是针对计算给定交易的交易净值这一任务而设计的DSL。显然这已经是以Java作为实现语言所能得到的最好结果了。站在程序员的角度，这段DSL很好理解，而且熟悉GoF设计模式（4.7节文献[1]）的程序员会很满意地看到抽象的模式原理被落实到一个具体的领域实现。不过，我们还能做得更好一些吗？

2. 改进Java实现

我们可以凭借Ruby或Groovy的反射式元编程能力提高DSL的表现力和动态性。不过，在查看具体的实现之前，我们先来确定一下哪些地方有改进的潜力。请看表4-1。

表4-1 先前用Java和装饰器模式实现的DSL可能具有的改进点

能否改进	如何改进
表现力和领域友好度	可以更简洁一些。毕竟对这方面的追求是无止境的
Trade抽象和装饰器被硬性捆绑在一起	去除静态继承关系，这可提高装饰器的重用性
易读性。Java实现阅读顺序由外而内，要穿过一长串装饰器才能看到核心的Trade抽象，不符合一般直觉	把Trade放在前面，装饰器放到后面

Ruby、Groovy等动态类型语言比Java的语法简练不少。Ruby和Groovy都具备鸭子类型（duck typing）特性，通过牺牲对于Java、Scala等语言开箱即用的静态类型安全，可以换取更有利于重用的抽象。（其实Scala也可以实现鸭子类型，详见第6章。）表4-1所列的改进点要求你对Ruby的动态性有更深入的了解，所以我们接下来介绍这方面的一些知识。

3. Ruby中的动态装饰器

代码清单4-6中的Ruby代码是和先前的Java实现差不多的`Trade`抽象，其中充实了一些较为贴近现实的内容。



Ruby知识点

- ❑ Ruby的模块（module）特性有利于实现mixin，mixin可以附加在其他类或模块上。
- ❑ 通过反射来生成运行时代码的相关Ruby元编程基础知识。

代码清单4-6　`Trade`抽象的Ruby实现

```
class Trade
  attr_accessor :ref_no, :account, :instrument, :principal

  def initialize(ref, acc, ins, prin)
    @ref_no = ref
    @account = acc
    @instrument = ins
    @principal = prin
  end

  def with(*args)
    args.inject(self) { |memo, val| memo.extend val }
  end

  def value
    @principal
  end
end
```

❶ 动态模块扩展

与之前的Java实现相比，只有`with`方法❶这一部分比较值得讨论，其他部分差异不大。我们等下再回头讨论`with`方法，现在先看看装饰器的部分。我把装饰器设计成Ruby模块的形式，使用的时候可以将其作为mixin混入到实现主体之中（关于mixin，请参阅A.3节）：

```
module TaxFee
  def value
    super + principal * 0.2
  end
end

module Commission
  def value
    super - principal * 0.1
  end
end
```

显而易见，这段代码中的装饰器并没有静态地耦合到作为抽象主干的Trade类。我们轻而易举地就达成了表4-1中的一项改进目标。

我们刚刚不是提过鸭子类型吗？上术代码片段中的这些模块可以混入到任意实现了value方法的Ruby类中；正好我们的Trade类就有这么一个value方法。那么上面模块定义中的super调用是怎么发挥作用的呢？在Ruby语言里，如果你指定了一个不带任何参数的super调用，Ruby会发送一条消息给当前对象的父对象，请求调用父对象的同名方法。这就是反射式元编程大显身手的时刻了。此处调用的是value方法。我们调用super类方法的时候并不需要静态地绑定任何具体的父类。图4-4展示了指向Trade类和各装饰器的super调用如何在运行时动态串联到一起，以达到我们所期望的效果。

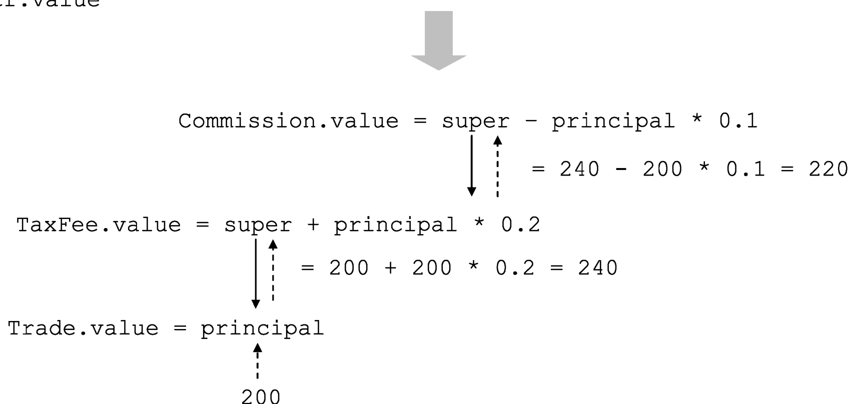


图4-4　展示super调用如何将value()方法串接在一起。调用从Commission.value()开始，Commission是链条中的最后一个模块，调用向下逐级传播，直到Trade类。实线箭头连接起来就是调用的链条；求值计算沿着虚线箭头依次进行，最终求得结果220

在这个例子里，元编程的神奇作用体现在什么地方？它怎样改善DSL的表现力？若回答这两个问题，我们要回头说说代码清单4-6中的with方法。这个方法的作用是把作为参数传递给它的所有装饰器动态地连接到Trade对象，使Trade对象扩展成为新的抽象。图4-5说明了装饰器与主体类动态组合的原理。

我们完全可以通过对Ruby类的静态扩展取得与图中动态扩展相同的效果。但是在运行时进行扩展更有利于各部分抽象提高可重用性，降低耦合程度。代码清单4-6所实现的Trade对象与装饰器结合的例子如下：

```
tr = Trade.new('r-123', 'a-123', 'i-123', 20000).with TaxFee, Commission
puts tr.value
```

我们运用元编程技巧在运行时绑定对象，结果得到上面代码片段中的DSL。它的阅读次序由内而外，符合自然的领域语法。语法中的非本质复杂性低于先前的Java版本，对于领域用户而言

表现力明显提高。（关于非本质复杂性的讨论，请参阅A.3.2节。）至此，表4-1所列的3项改进全部成功完成。这一路简直太顺利了。

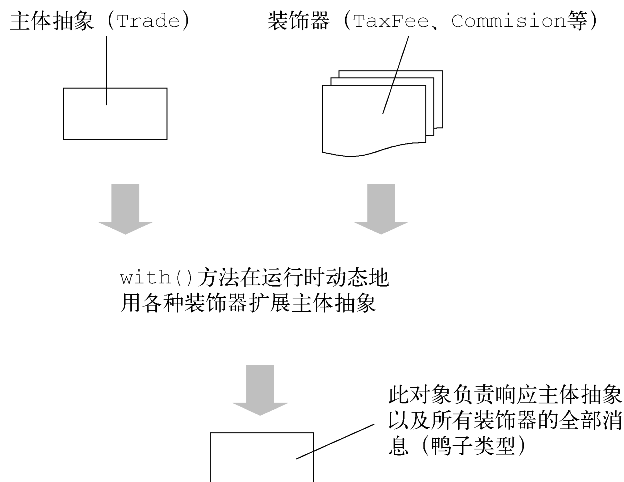


图4-5　主体抽象（Trade类）通过with()方法动态地与各种装饰器（TaxFee和Commission）连接起来，形成扩展

本节要点

装饰器设计模式用于在对象上附加额外的职责。如果能像本节用Ruby模块做到的那样把装饰器动态化，你将可以大大提高DSL的易读性。

不过，我们所面对的并非坦途。任何依赖于动态元编程的模式都有你必须小心应对的陷阱。请特别注意那些可能给日后造成麻烦的问题，参见下面的小小提醒。


在动态类型语言中使用运行时元编程将带给你更简洁的语法、更具表现力的领域语言，还有动态操控类结构的能力。而换取这些好处的代价是类型安全和运行速度两方面的牺牲。强调代价并不意味着劝阻你运用元编程技术设计DSL，只是提醒你不要忘记设计抽象的过程始终是一个权衡利弊的过程。在基于DSL的开发活动中，难免有时静态类型安全的重要性高于为DSL用户提供最佳表达手段的需要。本章后面还会讨论到，即使在满足静态类型检查的前提下，像Scala这样的语言仍然有办法让DSL的表现力不输于Ruby语言。所以千万别匆忙地下决定，先比较一下所有可能的方案吧。

本节介绍了如何利用元编程来实现动态的装饰器。其实现方式与Java、C#等静态类型语言差别很大，表现出了更高的灵活性。最重要的是，我们见识了动态装饰器用于实现DSL代码的真实案例。接下来，我们继续探索反射式元编程技术，把另一种常见的Java设计模式活用于DSL设计。

4.2.3 利用buider的反射式元编程

在第2章作为入门练习，我们实现过一个订单处理DSL，还记得吗？当时在Java版本的实现里面，我们为了提高DSL对于用户的表现力运用了Builder设计模式（4.7节文献[1]）。下面是从2.1.2节照搬过来的一段代码，订单处理DSL的Java实现使用起来是这样子的：

```
Order o =
  new Order.Builder()
    .buy(100, "IBM")
    .atLimitPrice(300)
    .allOrNone()
    .valueAs(new OrderValuerImpl())
    .build();
```

代码中运用了连贯接口这种常见手法去构建一个完整的Order对象。只是由于Java的缘故，整个构建过程是静态的；builder提供的所有构建方法全部只能静态地调用。如果借助Groovy语言的动态元编程模式，我们有办法提高builder的“极简”程度，同时又不削弱其表现力（4.7节文献[2]；关于抽象设计中的极简特质，请参阅A.2节）。这样用户不用写那么多八股代码，得出来的DSL比较精干而且更易于掌握。在静态方式下需要堆砌大量死板代码来完成的事情，语言运行时利用反射就做到了（所以叫反射式元编程）。



Groovy知识点

- Groovy中如何定义类和对象。
- Groovy builder允许你使用反射建立对象的层次结构，其语法简练，特别适合运用于DSL。

1. Groovy builder的魔力

代码清单4-7对运用Groovy对Trade对象进行了建模，从中可以看出它的基本组成要素。这里需要再次说明，我们举例所用的Trade抽象因应本章的议题作了大幅度的简化，不可以和真实交易系统中的情况相提并论。

代码清单4-7 Groovy实现的Trade抽象

```
package domain.trade

class Trade {
  String refNo
  Account account
  Instrument instrument
  List<Taxfee> taxfees = []
}

class Account {
  String no
  String name
  String type
}
```

```
class Instrument {
  String isin
  String type
  String name
}

class Taxfee {
  String taxId
  BigDecimal value
}
```

这是一个平平无奇的Trade抽象，它由一个Account对象、一个Instrument对象和一个Taxfee对象列表组成。我现在要介绍一段builder脚本，它能神奇地探知抽象内部的类结构，用你提供的值正确构造出抽象内的各个对象。

代码清单4-8　作用于Trade对象的动态builder，用Groovy语言编写

```
def builder =
  new ObjectGraphBuilder()

builder.classNameResolver = "domain.trade"
builder.classLoader = getClass().classLoader

def trd = builder.trade( refNo: 'TRD-123') {
  account(no: 'ACC-123', name: 'Joe Doe', type: 'TRADING')
  instrument(isin: 'INS-123', type: 'EQUITY', name: 'IBM Stock')
  3.times {
      taxfee(taxId: 'Tax ${it}', value: BigDecimal.valueOf(100))
  }
}

assert trd != null
assert trd.account.name == 'Joe Doe'
assert trd.instrument.isin == 'INS-123'
assert trd.taxfees.size == 3
```

❶ 建立builder

❷ 动态地产生方法

对于一直使用Java语言的开发者来说，上面的代码确实有点神奇。DSL用户在脚本中写了一个trd❶方法，它构造了一个创建交易对象的builder。在trd方法里面，用户调用了account、instrument❷等方法,可是这些方法明明Trade类里面从来没有定义过，就好像语言运行时把它们变出来的一样，代码居然能正确执行。这其实是Groovy元编程变的“小戏法”。

2. 揭开Groovy builder的秘密

除了元编程技术参与了“戏法”，命名参数和闭包也“出了很大力气”，才让代码清单4-8中的DSL脚本表现出那样奇妙的结果。前面各章的例子已经多次展示过闭包在Groovy语言中的用法。现在我们更深入一点，仔细看看语言运行时是怎样探知类名、正确创建实例，然后用builder获得的数据填充实例的属性的。要点参见表4-2。

表4-2　builder与元编程

运行时的作用	工作原理
匹配方法名	在ObjectGraphBuilder上调用任何方法，Groovy都会把方法名通过ClassNameResolver策略进行匹配，匹配到的Class对象就是将要实例化的类

（续）

运行时的作用	工作原理
设置ClassNameResolver	用户可以自定义ClassNameResolver策略，代之以自己的实现
创建实例	Groovy得到Class对象之后，会应用策略NewInstanceResolver，调用目标类的无参数构造器创建该类的一个默认实例
处理类结构和层次关系	如果目标类的内部引用了别的类，形成了父子关系（如代码清单4-8中的Trade和Account），builder会更复杂一些。遇到这样的情况，builder会应用RelationNameResolver、ChildPropertySetter等其他策略去确定属性所属的类，然后实例化它们

如果想进一步了解Groovy builder的工作细节，请参考4.7节文献[2]。

你已经看了不少元编程技术，对于怎样用它们设计具有表现力的DSL有了相当程度的了解。现在全面观察一下本节中我们所做过的事情，回顾图4-2中我们目前为止尝试过的所有模式。这些模式就是你今后闯荡DSL世界的工具了。

> **本节要点**
>
> **builder可以用于在DSL里面分步构造对象**。builder被动态化之后，大幅减少了不得不写的死板代码。Groovy或Ruby语言中的动态builder通过语言运行时的元对象协议动态地构造方法，大大缓解了DSL实现方面的负担。

4.2.4　经验总结：元编程模式

切不可将我们讨论过的众多模式视为一个个互不关联的实体。面对具体领域的时候，你会发现每一种模式在应用之后，都可能形成需要用另一种模式去化解的作用力（4.7节文献[5]）。图4-6简要回顾了本章到目前为止实现过的模式。图4-6将图4-2列举的DSL模式列于左侧，而本章举例讨论过的具体实现方案则列于右侧。

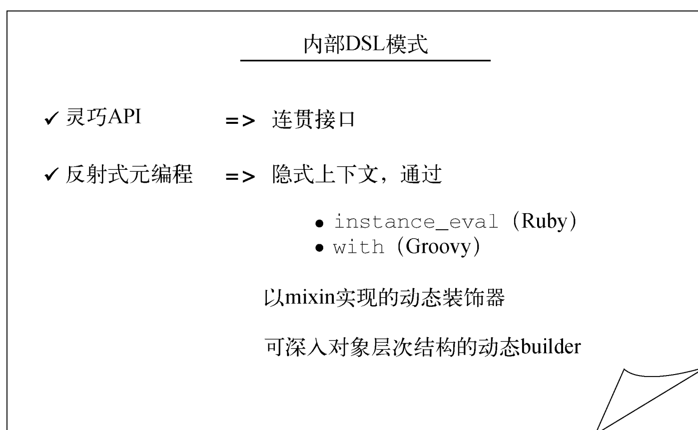


图4-6　目前为止介绍过的内部DSL模式。本章已经在Ruby和Groovy语言中实现了这些模式

我们讨论的几个模式都相当重要，实现内部DSL的时候你会常常用到。这些模式主要针对具备较强元编程能力的动态类型语言。

介绍完反射式元编程，我们即将探讨另一类模式，这类模式将用于在像Scala那样的静态类型语言里面实现内部DSL。当你用类型化的抽象来建模DSL元素时，语言类型系统中的一些规则可以自然地充当起领域中的业务规则。这也是一种保持DSL简练同时又不失其表现力的途径。

本节要点

本节讨论的模式可帮助你降低DSL的烦琐度，提高动态性。我们利用语言的元编程能力在运行时完成必要的工作，以此取代静态方式下那些死板的代码。

重要的不仅是具体的Ruby或Groovy语言实现，你需要全面理解塑造了这些实现的上下文环境。有些强大的实现语言能给你提供丰富的手段去创作动态的DSL。

当你用这里学到的技巧去解决实际的领域建模问题，从而对问题有了更深刻的体会，你将会给这些技巧找到更多的用武之地，也会创造出自己的解决之道。

4.3　内嵌式 DSL：类型化抽象模式

迄今为止，我们对于模式的探讨全都离不开精简DSL代码结构这个主题，而且从使用和实现两个方向进行了反复的讨论。

本节暂时把动态语言放到一边，我们试试看能否利用类型系统的威力激发DSL的表现力。本节的示例全部使用Scala语言（以Scala 2.8为准）。我们重点说明类型怎样（甚至在程序运行之前）给DSL的一致性增加一层额外保障。此外，类型在使DSL语言精练方面的能力不输于我们先前讨论的一些动态语言。多看看图4-2，本章讨论的所有模式都在那个大纲里面。

4.3.1　运用高阶函数使抽象泛化

一直以来凡是涉及领域的讨论，我们总是拿金融中介系统里的操作来举例，如维护客户账户、处理交易和成交、代客户下单等。本节我们来看一份客户文件，上面记录了中介在工作日内进行的所有交易活动。交易组织会为某些客户生成这样一份账户每日交易活动报表，然后发送到客户的邮件地址。

1. 生成一份分组报表

图4-7是一份账户活动报表，里面记录了交易的票据品种、数量、时间、金额。

很多组织会为客户提供灵活的每日交易情况查看方式。客户可以要求交易情况按某一项表格元素排序或者分组。比如，我会希望一天内所有的交易按照票据品种排序并且分组显示，如图4-8所示。

我也可以要求把所有的交易按照交易数量来分组，如图4-9所示。

账户名称：______	地址：______		
2009年12月12日的交易活动			
票据	数量	时间	金额
Google	2000	08:20	....
IBM	1200	11:30	
Google	350	11:45	
Verizon	350	12:10	
IBM	2100	12:20	
Google	1200	12:50	
....	....	....	

图4-7　经过简化的账户活动明细报表

账户名称：______	地址：______		
2009年12月12日的交易活动：			
票据	数量	时间	金额
Google	2000 1200 350	08:20 11:45 12:50	....
IBM	1200 2100	11:30 12:20	
Verizon	350	12:10	
....	....	....	

图4-8　一份账户活动报表，按照票据品种进行了排序和分组。注意票据栏的排序方式，数量栏将同一种票据的记录排在了一起

现实中的账户活动报表还会有其他很多内容，不过图中的信息已经足够满足下面的实现和讨论需要了。我们现在要构建一种DSL来实现自定义的分组函数，让客户按需要调整交易活动报表的样式。一开始，我们可以这样设计DSL：每种分组操作分别用一个函数来实现。然后，我们考虑设计一个泛化的`groupBy`组合子——即一个以分组条件为参数的高阶函数——来改善DSL的精炼度。

账户名称：＿＿＿＿＿＿　　　地址：＿＿＿＿＿＿＿

2009年12月12日的交易活动

数量	票据	时间	金额
350	Google Verizon	12:50 12:10	
1200	Google IBM	11:45 11:30	....
2000	Google	08:20	
2100	IBM	12:20	
....	....	.....	

图4-9　一份账户交易报表，按照当日的交易数量进行了排序和分组

定义　组合子是以其他函数作为输入的高阶函数。组合子可以被组织起来构成DSL的语言结构，本节以及第6章都有这方面的例子。附录A也详细讨论了组合子。

Scala类型系统可以保证操作的静态类型安全，又有着处理高阶函数的能力，你阅读完本节的例子，必定会对Scala的这些特点深有体会。那么，我们就直接来看代码示例吧。



Scala知识点

- **case类定义不可变的值对象**。case类是一种简洁的抽象设计手段，可以用于自动获得编译器提供的许多额外便利。
- 针对已有的抽象，**隐式类型转换**提供了一种完全非侵入的扩展方式。
- **for-comprehension**特性针对集合上的迭代子提供了一种函数式抽象。
- 运用**高阶函数**，你可以设计和组织起能力强悍的函数式抽象。

2. 建立基本抽象

我们来试一下从后往前推的方法，先设想一下DSL最后的样子，再尝试用Scala实现出来。用户将会这样使用我们的DSL：

```
activityReport groupBy(_.instrument)
activityReport groupBy(_.quantity)
```

第一行DSL代码生成一份按票据品种分组的活动报表，第二行代码生成的报表则按交易数量分组。下面的代码片段实现了账户活动报表的基本抽象，我们来仔细看看它具备的一些特性。

```
type Instrument = String                                    ❶ 具体类型定义

case class TradedQuantity(instrument: Instrument,
  quantity: Int)                                            ❷ 值对象

implicit def tuple2ToLineItem(t: (Instrument, Int)) =
  TradedQuantity(t._1, t._2)                                ❸ Tuple2到LineItem的隐式类型转换

case class ActivityReport(account: String,
  quantities: List[TradedQuantity]) {                       ❹ 主体抽象
  //..
}
```

本书不是一本Scala专著，但为了帮助读者理解本书，我将重点说明这段代码展现出来的一些语言特性。这样一来，你可以把它和等价的Java代码相比较，从而对它的表现力水平有一些直观的认识。

实现DSL随时都要注意对表现力的要求。代码开头用一则类型定义来对领域制品建模❶，避免直接套用意义含糊的原生数据类型，让代码直接说明自身的含义。这样不但利于领域用户的理解，还给`Instrument`类型留下了日后修改的余地。

`TradedQuantity`❷是个case类，它建模了一个值对象。值对象一般被认为是不可变的，选用Scala的case类作为表达手段正是用其所长。case类的特点是数据成员自动具备不可变性质，拥有特别简便的内建构造器语法，而且默认实现了`equals`、`hashCode`和`toString`方法。（Scala语言的case类特别适合用于建模值对象。详细介绍请参阅4.7节文献[4]。）

Scala语言通过隐式声明❸提供数据类型之间的自动转换。Scala的隐式特性是一种限定了词法作用域的语言结构，也就是说，只有当一个模块明确导入了隐式定义，类型转换才在该模块范围内生效。在本例中，按照声明，二元组`(Instrument, Int)`可被这种声明隐式转换成一个`TradedQuantity`对象。Scala允许用`(Instrument, Int)`这种字面写法来表示二元组，它的完整写法是`Tuple2[Instrument, Int]`。（前面3.2.2节讨论过Scala语言`implicits`特性的工作原理，如有需要可以翻回去温习一下。）

最后说到账户活动报表的主体抽象。`ActivityReport`❹包含账户信息和当日所有交易活动的一个列表，列表元素是交易数量和交易品种组成的二元组。

接下来我们就要展开一系列迭代式的建模过程，实现分组函数，满足客户自定义每日交易报表显示方式的需要。我们打算用迭代式的过程逐步改进模型，见表4-3。

表4-3　对DSL的迭代式改进

步　骤	说　明
创造一种DSL供用户查看交易活动报表，该语言有按照`Instrument`和`Quantity`进行分组的功能	每种分组条件各实现一个专门的分组函数，即`groupByInstrument`和`groupByQuantity` 实现泛型分组函数`groupBy[T <% Ordered[T]]`，以减少重复性的固定代码

那么，我们先来实现几个`groupBy`函数。

3. 第一步：专用实现

如果我们在`ActivityReport`抽象内按分组条件提供专用的`groupBy`函数，DSL用户得到的

API表现力会很好。不过我们还要从实现的角度去考虑，使用方面的表现力并非判定DSL完善程度的唯一标准。代码清单4-9给出了按照`Instrument`和`Quantity`分组的专用实现。注意每种分组条件都需要单独定义一个专用函数。

代码清单4-9　活动报表，groupBy函数采取专用实现

```
type Instrument = String

case class TradedQuantity(instrument: Instrument,
  quantity: Int)

implicit def tuple2ToLineItem(t: (Instrument, Int)) =
  TradedQuantity(t._1, t._2)

case class ActivityReport(account: String,
  quantities: List[TradedQuantity]) {
  import scala.collection.mutable._

  def groupByInstrument = {
    val m =
      new HashMap[Instrument, Set[TradedQuantity]]
        with MultiMap[Instrument, TradedQuantity]          ❶ 用mixin方式定义MultiMap

    for(q <- quantities)
      m addBinding (q.instrument, q)                       ❷ for comprehension

    m.keys.toList
          .sortWith(_ < _)
          .map(m.andThen(_.toList))                        ❸ 按票据品种分组
  }

  def groupByQuantity = {
    val m =
      new HashMap[Int, Set[TradedQuantity]]
        with MultiMap[Int, TradedQuantity]

    for(q <- quantities)
      m addBinding (q.quantity, q)

    m.keys.toList
          .sortWith(_ < _)
          .map(m.andThen(_.toList))
    }
  }
```

你能看出这种实现方案的缺点吗？让我们先简单看下代码中用到的一些Scala惯用法，然后再作进一步的分析。

在代码清单4-9的`ActivityReport`实现里面，`quantities`可以含有对应同一个`Instrument`对象的多个条目，所以我们定义一个`MultiMap`容器❶来归置从`quantities`取出的条目，定义`MultiMap`容器用到Scala的mixin语法。在`HashMap`对象上我们混入trait `MultiMap`，就得到`MultiMap`的具体实例。关于Scala语言中trait和mixin特性的详细解释，请参阅4.7节文献[4]。

在遍历`quantities`并填充`HashMap`的时候，我们运用了Scala语言的`for` comprehension特性❷。它和命令式语言中的`for`循环有明显区别（6.9节谈及Scala语言的Monad化结构的时候，我们

再详细讨论for comprehension特性）。然后，我们对MultiMap的键进行排序并建立一个按Instrument分组的List❸。该List的每个元素都是一个Set容器，里面存放了某票据品种对应的全部交易数量条目。代码中下划线的语法含义与3.2.2节介绍的相同。

代码清单4-9中实现的主要缺点是代码重复部分较多。groupByInstrument和groupByQuantity函数在结构上完全相同，只有作为分组依据的属性不一样。你应该马上就警觉到，这种情形违反了优秀抽象的设计原则。万一你还没有认识到其中的缺点，请翻阅附录A，那里介绍了如何对抽象进行精炼以摒除非本质复杂性。总之，专用实现会助长重复性代码，这就是问题的症结。而且，如果日后向ActivityReport类增加更多分组条件，那些刻板代码只会一再重复出现。怎样才能纠正当前实现的缺点呢？我们需要更一般化的实现方案。

4. 一般化的实现

我们现在就把实现推广到更一般化的情况，将原来分立的一系列专用方法概括成一个通用的方法。

代码清单4-10　泛型groupBy实现

```
type Instrument = String

case class TradedQuantity(instrument: Instrument,
  quantity: Int)

implicit def tuple2ToLineItem(t: (Instrument, Int)) =
  TradedQuantity(t._1, t._2)

case class ActivityReport(account: String,
  quantities: List[TradedQuantity]) {
  import scala.collection.mutable._

  def groupBy[T <% Ordered[T]](f: TradedQuantity => T) = {    ❶ 把分组条件参数化
    val m =
      new HashMap[T, Set[TradedQuantity]]
        with MultiMap[T, TradedQuantity]
    for(q <- quantities)
      m addBinding (f(q), q)
    m.keys.toList.sort(_ < _).map(m.andThen(_.toList))
  }
}
```

新的实现明显更简短。你有没有发现，泛型groupBy❶方法产生了更有力的抽象，同时代码的紧致度也在同步上升。表4-4简要总结如何实现泛型groupBy方法。

表4-4　实现泛型groupBy方法

步　　骤	说　　明
实现泛型groupBy方法	将groupBy方法参数化，带上对活动报表分组所依据的类型
	groupBy方法接受f函数作为输入参数，f对分组条件建模。这里体现了Scala对高阶函数的支持。函数可以像其他数据类型一样被当做参数和返回类型传来传去。对分组条件进行抽象的时候可以利用这个特点，代替代码清单4-9中的专用实现
	图4-10可以帮助你理解泛型groupBy函数的执行过程和原理

下面我们来观察DSL用户调用groupBy的过程，把实现步骤从头到尾过一遍。这样的练习可以帮助你理解Scala的类型系统是怎样在背后发挥作用，默默塑造出富于表现力的DSL语言结构的。

探究现象背后的来龙去脉是DSL设计工作的重要一环，DSL的实现者尤其应该全面、细致地掌握相关知识。读者有必要反复阅读本节，直到完全理解Scala语言如何进行方法分发。代码清单4-10中短短15行的实现代码里面隐藏了相当数量的惯用法，值得你好好学习。当你能够看清各种惯用法之间互相联系的脉络，知道怎样将它们契合在一起满足API的契约，这些惯用法就会成为你手中的“工具”。

```
val activityReport =
  ActivityReport("john doe",
    List(("IBM", 1200), ("GOOGLE ", 2000), ("GOOGLE", 350),
      ("VERIZON", 350), ("IBM", 2100), ("GOOGLE", 1200)))

println(activityReport groupBy(_.instrument))
println(activityReport groupBy(_.quantity))
```

与其用文字说明上面的代码，我们不如用图4-10来解释activityReport groupBy (_.instrument)调用前后发生的一系列动作。

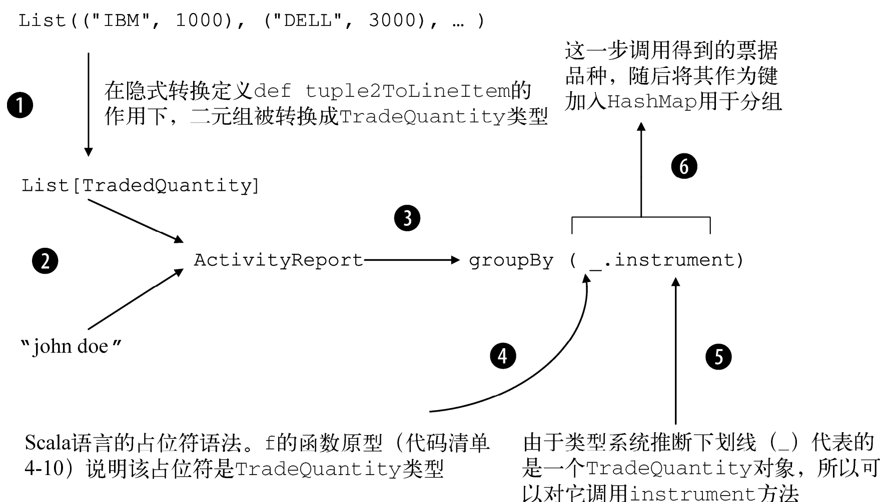


图4-10　按票据品种分组的活动报表（groupBy(_.instrument)）的产生过程。顺序观察图中所有步骤，将图解与代码清单4-10的DSL实现，以及上文运用DSL为客户john doe生成ActivityReport的代码片段相对照

高阶函数的应用场合远不止本节所介绍的类型化抽象模式。所有现代语言，无论是否静态类型的语言，全都支持高阶函数和闭包。本节讨论的模式实现可以套用到不同的情形和语言中去。实践者要留心使用模式的上下文环境，善用实现语言的特点把事情做好。

我们的目标是跨越实现语言的界限，探索JVM平台上所有的内部DSL实现模式。无论你选择静态类型还是动态类型的实现语言，总之应该根据DSL建模所需的能力去挑选合适的工具。

下一节将讨论怎样运用显式类型约束来表达领域逻辑和行为。这种表达手段不适用于动态类型语言。不过只要类型系统表现力充沛，显式类型约束可以成为得力的建模工具。它对于使DSL语言简洁有着惊人的效果。

4.3.2 运用显式类型约束建模领域逻辑

设计领域模型的时候，抽象必须遵照领域所施予的规则和约束去实现其行为。Ruby、Groovy等动态类型语言将领域规则表达为运行时的约束。4.2节已经演示过Ruby和Groovy语言的反射式元编程手法，讲解过怎样实现一些DSL语言结构来建立领域规则的模型。本节将首先用Ruby语言实现一个运行时验证示例，然后演示如何借助Scala语言的静态类型系统更简洁地表达类似约束。

1. Ruby语言的运行时验证

我们继续沿用代码清单4-6中的`Trade`抽象这个例子，先前已经用Ruby语言建立一个简单的领域模型。`Trade`对象需要对应一个`Account`对象，即客户的*交易账户*。代码清单4-6把账户对象表示为类方法`attr_accessor`。交易系统的领域概念里存在很多不同类型的账户（参见3.2.2节的补充内容“金融中介系统：客户账户”）。对于`Trade`抽象来说，这个账户被限定为*交易账户*，*结算账户*不可以用在这个地方。那么，我们每次建立`Trade`对象并为它设置`Account`对象的时候，都必须验证这条领域规则。用Ruby语言应该怎么写呢？你可以像下面的代码片段一样插入常用的检查语句：

```
class Trade
  attr_accessor :ref_no, :account, :instrument, :principal

  def initialize(ref, acc, ins, prin)
    @ref_no = ref
    raise ArgumentError.new("必须为交易账户")
      unless trading?(acc)
    @account = acc
    ## ..
```

凡是领域模型中要求接受交易账户的地方，你都要在*运行时*反复执行同样的验证。（我们可以采取类似Rails的做法，把验证操作写成类方法，实现声明式的验证，但验证操作终究还是在运行时进行的。）而且每一处验证都必须明确地进行单元测试，确认当输入非交易账户的时候领域行为如同预期的一样中止执行。以上重重防范必然要增加代码量，但如果语言允许显式规定类型化的约束条件，我们就可以节省这部分代码。

在静态类型语言里，我们可以把约束条件用类型的方式规定出来，让它们在编译时接受编译器的检查。如果一个程序能正确编译，那就说明模型中领域行为的一致性至少有了一重保证。

2. Scala语言的显式类型约束

我们尝试用Scala语言建模`Trade`对象，对其中的账户和票据加上一些具有领域含义的约束条件。经过本例的练习，你将认识到在显式类型约束的作用下，DSL抽象无需实际运行就已经得到了一层额外的一致性保证，而这是动态类型语言所不具备的。假如你选择静态类型语言作为实现语言，那么显式类型约束绝对是不可或缺的一样工具。



Scala知识点

- **基于类型的编程方法**。以类型为手段，在DSL中表达领域约束。泛型类型参数和抽象类型都是你的好帮手。
- **抽象的`val`**成员可使抽象在进入最后的实例化阶段之前保持开放。

每个`Trade`对象都对应一个`Trading`账户。我们用Scala语言对这条规则进行建模。代码清单4-11只展示了`Trade`对象中与本段讨论相关的一个侧面。

代码清单4-11　带上类型化约束的`Trade`对象，Scala语言

```
trait Account
trait Trading extends Account
trait Settlement extends Account
trait Trade {
  type A <: Trading

  val account: A
  def valueOf: Unit
}
```

❶ 两种`Account`类型
❷ `Trading`子类型的账户
❸ `Account`实例

从这个示例中我们可看出类型如何隐式落实业务规则。这段代码用了Scala语言的trait特性建模`Account`和`Trade`对象（参见4.7节文献[4]）。我们为`Trading`账户和`Settlement`账户各安排了一种类型❶。程序员不可以向要求`Trading`账户的方法传递`Settlement`账户。编译器会捍卫这条规则，要求`Trading`账户的相关业务规则不需要特别去检测账户类型是否符合要求。

有一些业务规则是在代码中明确规定的。我们在`Trade`的定义里对抽象类型`A`设置了约束（`<: Trading`）❷，因此不能使用`Trading`之外的任何账户类型来实例化`Trade`对象❸。用户不需要另外增加验证账户类型的代码，这条规则的检验工作同样由编译器代劳。

“交易”是指参与票据买卖双方之间订立的合约。如果想复习交易的一些性质，请回头翻阅1.4节的补充内容。根据交易的票据种类的不同，交易的行为、周期过程和计算方法都有差异。股权交易（equity trade）涉及股票与现金的交换。而当被交换的票据属于固定收益类型，我们称之为固定收益交易（fixed income trade）。关于股权、固定收益等票据类型的详情，请阅读本节的补充内容。



金融中介系统：票据类型

票据的类型可说是五花八门，而它们都是为了迎合投资者和发行者的需要而设计的。类型不同，票据的交易、结算过程的生命周期也不同。

票据主要分为股权（equity）和固定收益（fixed income）两大类。

股权类票据又可进一步分为普通股、优先股、累积股、权证、存托凭证。固定收益类证券（又称债券）包括直接债券、零息债券、浮动利率债券。就我们的讨论而言，并无必要全面了解这方面的详细内容，我们只要记住当交易的票据类型不同，对应的`Trade`抽象也不一样即可。

下面的代码将Trade抽象的定义进一步具化，建立EquityTrade和FixedIncomeTrade的模型。

代码清单4-12　EquityTrade和FixedIncomeTrade模型

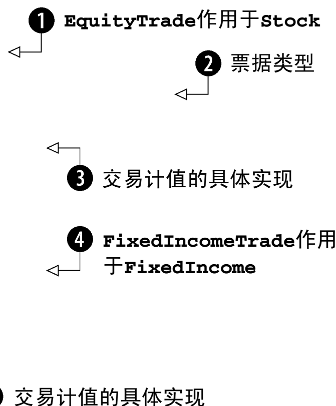
```
trait Instrument
trait Stock extends Instrument
trait FixedIncome extends Instrument

trait EquityTrade extends Trade {
  type S <: Stock

  val equity: S
  def valueOf {
    //..
  }
}

trait FixedIncomeTrade extends Trade {
  type FI <: FixedIncome

  val fi: FI
  def valueOf {
    //..
  }
}
```



代码中对所交易票据的类型进行了约束，类似于代码清单4-11中对Account所做的显式约束。此业务规则照旧由编译器隐式地强制实施。

我们分别规定了EquityTrade类型❶和FixedIncomeTrade类型❹。程序员不可以把FixedIncomeTrade对象传递给要求EquityTrade对象的方法。编译器会捍卫此规则，对具体的Trade类型有所要求的业务规则不需要特别去检测交易类型是否符合要求。

EquityTrade负责处理Stock交易❶，FixedIncomeTrade负责FixedIncome交易❹。我们据此分别对抽象val（equity❷和fi❺）进行了约束。以上基本业务规则完全在编译器层面得到保证，程序员无需编写一行验证代码。

valueOf方法是多态的、类型化的。不同的Account和Instrument类型对应着不同的Trade抽象，也分别对应着不同的valueOf方法实现（❸和❻）。

我们运用类型化抽象手段，并且对值和类型施加显式约束，在没有写一行过程式逻辑的情况下成功描述了相当数量的领域行为。不仅代码规模缩小了，单元测试的数量也减少了，直接降低了编写和维护的负担。当你维护代码的时候，如果类型标注能描述性地说明模型背后的领域含义，那么维护工作不是会顺利得多吗？

对比之前关于用动态语言实现DSL的讨论，本节讨论中体现出来的实现思路很不一样。现在我们总结一下什么是*静态类型思维*，看它和Ruby或Groovy的思维方式有何区别。

4.3.3　经验总结：类型思维

经过本节的学习，你已经知道对于设计表现力丰富的领域抽象，类型可以起到很重要的作用。

由于拥有静态类型检查这张安全网，静态类型的实现天生就具备一层正确性保障，这就是它与前面的Groovy和Ruby示例的主要区别。类型化的代码只要能通过编译，就足矣证明它满足了相当数量的领域约束。在第6章用Scala设计更多DSL的时候，我们会继续讨论这个议题。图4-11总结了本节介绍过的几种内部DSL模式。

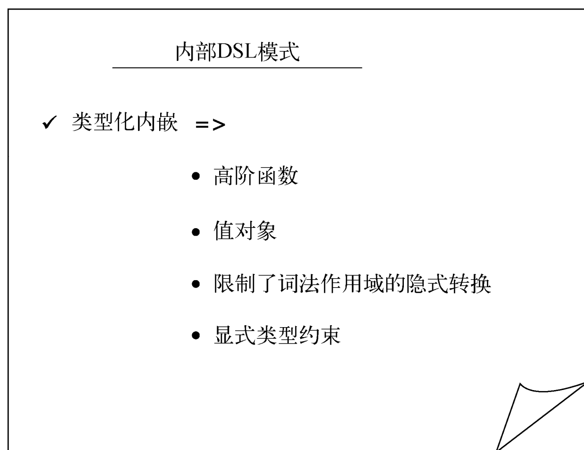


图4-11　类型化内嵌方式下用于内部DSL的程序结构。你可以从这些模式中学会运用类型思维去驾驭编程语言

我们讨论了运用静态类型语言实现内部DSL的过程中常会使用的若干重要模式。虽然静态类型语言没有动态语言那样的元编程绝技，但类型化抽象同样是非常简洁有力的DSL开发手段。

本节要点

本节的主要目的是引导你**用类型思考**。对于领域模型中的每个抽象，你应该把它类型化，然后围绕类型组织相关的业务规则。很多业务规则会自动被编译器强制实施，因此你不需要专门为之编写代码。如果实现语言拥有合适的类型系统，那么DSL的简洁程度不会亚于动态语言的实现。

到目前为止，我们介绍了不少内部DSL实现模式，有利用类型系统抽象出领域规则的，也有利用宿主语言的元编程能力做反射的。下一节将要介绍的模式能让语言运行时帮你编写代码。你要用生成的代码来创作简洁的DSL。

4.4　生成式 DSL：通过模板进行运行时代码生成

元编程有许多侧面，例如上文展示过不少反射式元编程的例子。VM在运行时对各种元对象进行探测，找出当前上下文内可用的对象，然后神奇地调用它们。我们还可以从不同的角度来看待元编程。其实，元编程最经典的定义是：编写“编写代码”的代码。

在不同的语言里面，这个定义的确切含义也不一样。Lisp等语言提供编译时元编程能力，我们在2.5.2节已经见识过。Ruby、Groovy等语言提供运行时元编程能力，可以在运行时通过`eval`和方法的动态分发生成代码。本节将用一个具体的例子向你展示如何减少直接编写的代码，转而依靠语言运行时生成余下的部分，以此达到使DSL抽象表面紧凑的目的。你可能会问，这种做法为什么很有意义？

4.4.1　生成式DSL的工作原理

设计生成式的DSL可以少写死板重复的代码。语言将通过元编程手段代替你生成代码。图4-12形象地说明了运行时元编程生成代码的情况。

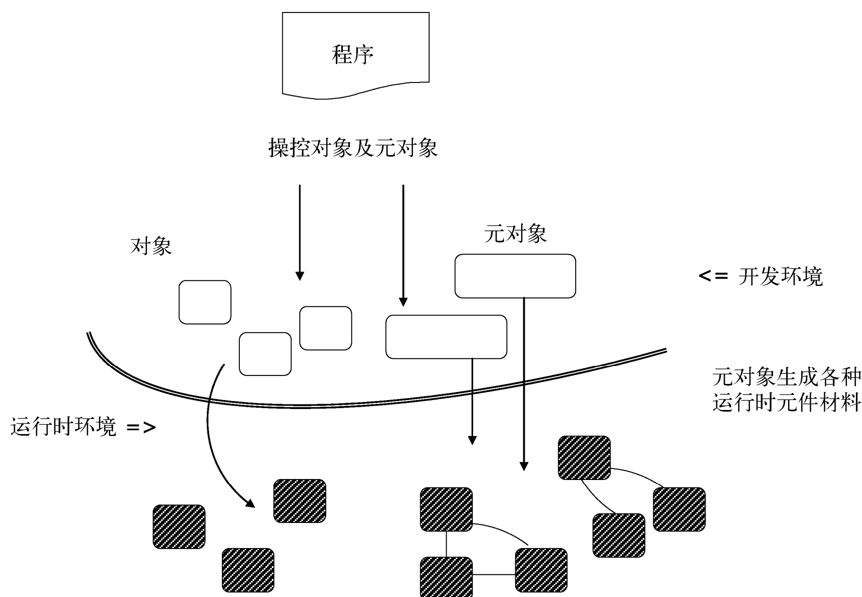


图4-12　运行时元编程方法在运行时根据元对象产生代码。元对象生成更多对象，结果是减少了死板代码的数量

程序员除了直接编写一部分对象，还操控元对象在程序运行的时候产生更多的程序元件。这些生成的元件材料就相当于语言运行时帮你编写的代码。这就好比你为了让自己集中精力对付更重要的工作而精明地指使助手，让他按照你的指令照顾好所有的例行工作。那么元对象具体是什么样子？它们在什么地方？它们怎样完成你的要求？我们一起用Ruby语言设计一些生成式DSL，边做边解释。

4.4.2　利用Ruby元编程实现简洁的DSL设计

关于证券交易和结算领域，本章已经围绕“交易”谈了不少内容。在现实的应用程序中，`Trade`

是一个非常复杂的模块，含有大量的领域对象和相关的业务规则、约束条件、验证逻辑。其中一些验证逻辑是通用的，适用于同一上下文内的所有相似属性，而另外一些验证逻辑专用于特定的上下文，需要在类定义中明确地写出来。但不管哪一种验证逻辑，其一般的执行过程是相同的，可以集中在一起，也可以通过合适的技术手段统一生成。

我们来看看Ruby元编程在这方面有什么办法。

1. 用类方法对验证逻辑进行抽象

假设你正在用Rails开发交易程序，其中用了`ActiveRecord`进行持久化。那么按照一般惯例，`Trade`模型中会出现这样的代码片段：

```
class Trade < ActiveRecord::Base
  has_one                  :ref_no
  has_one                  :account
  has_one                  :instrument
  has_one                  :currency
  has_many                 :tax_fees

  ## ..

  validates_presence_of    :account, :instrument, :currency
  validates_uniqueness_of  :ref_no

  ## ..
end
```

如果你有过开发Rails项目的经验，肯定知道上面类定义中最后两行的作用。这两个Ruby类方法（`validates_presence_of`和`validates_uniqueness_of`）封装了一些针对属性的验证逻辑，设置属性的时候，传进去的参数要经过它们的检查。注意看那些作用于属性的领域约束，它们被干净利落地从公开的API界面上剥离出来，很好地示范了什么是精炼的模型设计。关于抽象设计的精炼原则，请翻阅附录A.3。在运行时，这些方法会生成相应的代码片段去验证各属性。



Ruby知识点

- **Ruby元编程**基础知识。Ruby的对象模型包含许多可以用于反射式和生成式元编程的元件材料，例如类、对象、实例方法、类方法、单例方法等。Ruby元编程机制允许你在运行时探查其对象模型，也允许动态地改变行为或生成代码。
- **模块**，以及通过mixin来扩展现有抽象的时候，模块在其中所起的作用。

2. 用mixin动态生成方法

我们之前在代码清单4-6中用mixin手法设计过`Trade`抽象，现在做点儿类似的事情。我们希望在`Trade`的类定义内写入内联的验证逻辑，但同时希望把调用和异常报告方面的烦琐构造隐藏起来，把那些重复出现的死板代码推到运行时去生成。完成后的抽象应该是下面的样子：

```
class Trade
  include ...

  attr_accessor :ref_no, :account, :instrument
  trd_validate :principal do |val|
    val > 100
```



❶ **include**什么？



❷ 验证逻辑以块的形式出现

```
  end

  ## ..
end
```

这段代码在❶的位置还少了些东西（我很快会说明少了什么）。trd_validate就是负责验证的“装置”，它要对以块的形式传入的验证逻辑❷生成运行时调用代码。

可是trd_validate是从哪里来的呢？我们必然要在别的什么地方定义这个方法，然后把它和Trade类的主体代码联系起来。答案也许就隐藏在❶省略的部分，我们把代码再看清楚一点儿。先不管Trade模型怎么获得trd_validate方法，先来看定义类方法trd_validate的Ruby模块TradeClassMethods：

```
module TradeClassMethods
  def trd_validate(attribute, &check)
    define_method "#{attribute}=" do |val|          ❶ 为属性生成设置方法
      raise 'Validation failed' unless check.call(val)
      instance_variable_set("@#{attribute}", val)
    end

    define_method attribute do                      ❷ 为属性生成获取方法
      instance_variable_get "@#{attribute}"
    end
  end
end
```

这段代码做了哪些事情？它利用Ruby语言在运行时动态定义方法的能力，为传递给trd_validate方法的属性生成setter❶和getter❷方法，此外还顺便生成代码去调用用户以块的形式传入的验证逻辑。真不错！想想这种元编程手段给每一次调用trd_validate省下了多少代码，再乘以所有需要trd_validate经手的属性数量，这一代码量相当可观。

3. 最后组装

一切就绪，到了最后组装起来的时刻。我们再定义一个模块把TradeClassMethods模块和Trade类粘合在一起，使trd_validate成为Trade的一部分。代码清单4-13起到最后画龙点睛的作用。

代码清单4-13　含有领域验证的Trade抽象

```
## enable_trade_validation.rb
require 'trade_class_methods'
module EnableTradeValidation
  def self.included(base)
    base.extend TradeClassMethods
  end
end

## trade.rb
require 'trade_class_methods'
require 'enable_trade_validation'

class Trade
  include EnableTradeValidation

  attr_accessor :ref_no, :account, :instrument
```

```
    trd_validate :principal do |val|
      val > 100
    end

    ## ..
  end
```

本例至此大功告成。为了避免直接编写验证逻辑的重复劳动，我们亲身体验了一把运用元编程手段生成验证逻辑代码的活动。注意，代码是在Ruby VM执行程序的时候，也就是*运行时*生成的。

本节要点

本章讨论的大多数模式重点放在降低DSL的烦琐程度，同时提高表现力上。Ruby和Groovy具有很强的运行时元编程和代码生成能力。当你发现DSL的实现代码显露出重复的迹象，请掂量一下要不要打开元编程的锦囊。与其自己写那些死板代码，不如让语言运行时帮你写。

本章虽长，但绝不沉闷。我们一直在演练各式绝招，将来你动手编写DSL的时候肯定会派上用场。看第一遍的时候感觉没学到家也不要紧，当你掌握了这些技巧背后的总体思路，就能看透问题的实质，用最恰当的方式刻画解答域。现在不妨换换脑子，在编程能力方面给自己充下电，因为下面即将讨论一种古老的程序开发范式在JVM平台上的新发展。我们要说的是Clojure——改头换面出现在JVM上的Lisp元编程。Ruby和Groovy元编程主要基于运行时代码生成，而Clojure元编程是通过宏在编译时完成的。我们将探讨宏会给内部DSL带来怎样的设计思路。

4.5 生成式 DSL：通过宏进行编译时代码生成

歇好了吧？那我们就开始吧。Ruby和Groovy的生成式元编程在运行时产生代码，使DSL表面语法保持紧凑，让语言运行时代替你编写那些死板代码。而对于Clojure（JVM上的Lisp），一方面你会获得所有代码生成方面的好处，另一方面它又没有Groovy和Ruby那种运行时负担。（关于Clojure语言的详情，请访问http://clojure.org。）Clojure语言按照其创造者Rich Hickey的设计，拥有Lisp语言的语法和语义，同时又能无缝地集成Java的对象系统。关于这种语言及其运行时的详情，请参阅4.7节文献[3]。

4.5.1 开展Clojure元编程

Clojure是一种动态类型语言，实现了“鸭子类型”，还提供强大的函数式编程能力。本节我们重点讨论它通过“宏”机制提供的代码生成能力。

Clojure通过宏来进行的代码生成属于一种*编译时元编程*，我们在2.3.1节提到过。如果你不熟悉Lisp或Clojure编译时宏的工作原理和基本概念，请阅读附录B。图4-13简要概括了编译时元编程系统的事件流程。开发者在程序中以宏的形式定义高阶抽象，然后高阶抽象在编译阶段被展开成正式的Clojure代码成分。

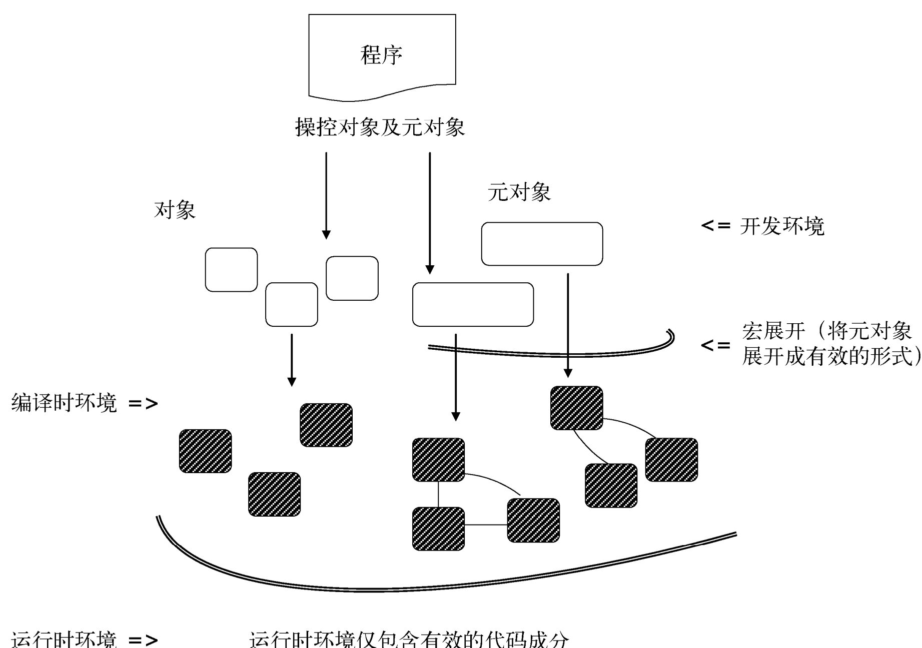


图4-13　编译时元编程通过宏展开的方式产生代码。注意代码产生的时间是在编译阶段，因此没有任何额外的运行时开销，与图4-12的情况不同

具体的实现细节我们等一下再谈，现在先对下面将要涉及的问题域做一点分析。当客户委托中介交易（无论买入还是卖出）某一品种的票据时，将依次发生以下动作：

(1) 中介向交易所提请交易；

(2) 各中介按照委托内容完成中介间交易，引发成交；

(3) 成交结果被分配到各客户账户，产生客户交易。

我们试着实现从成交结果产生客户交易的分配过程。在现实中，委托、成交结果、客户交易之间是多对多关系。为了让例子简单一些，我们假设成交结果与客户交易之间是一对一的关系。Clojure的鸭子类型将在这个用例的建模过程中贡献力量。

 Clojure知识点

- **前缀语法和函数式思维**。Clojure的语法建立在“S表达式”的基础上，采用前缀表示法（prefix notation）。Clojure的语法非常规则，标榜绝对一致性，例如没有运算符优先级。Clojure是函数式语言，各种模块被组织成函数的形式。
- **Clojure的map数据结构**普遍用于对象建模。
- **宏**可用于定义DSL的语法扩展。将宏在编译时生成代码的能力应用到DSL设计中，有利于使DSL语句简洁。

4.5.2　实现领域模型

我们来思考如何用Clojure定义交易与成交结果。交易和成交结果大致含有相同的信息，区别在于成交结果包含一个中介账户，而交易是在下单客户的客户账户上发生的。下面的代码片段分别给出了交易和成交结果的一个示例：

```
(def tr1
  {:ref-no "tr-123"
   :account {:no "cl-a1" :name "john doe" :type ::trading}
   :instrument "eq-123" :value 1000})

(def ex1
  {:ref-no "er-123"
   :account {:no "br-a1" :name "j p morgan" :type ::trading}
   :instrument "eq-123" :value 1000})
```

Clojure关键字 **::trading** ❶

在交易的数据结构之内，有一个独立的账户数据结构。账户含有一个`:type`属性，表明它是交易账户还是结算账户。该账户类型属性被建模为Clojure关键字的形式❶，只起到一个符号标识的作用，求值后等于自身。Clojure关键字提供快速的相等性测试，被当做轻量级的常量字符串来使用。关于客户账户的概念及其不同类型，请阅读3.2.2节的补充内容“金融中介系统：客户账户”。

我们在代码清单4-14中定义两个函数：其一检查给定的账户是否为交易账户，其二分配成交结果到一个客户账户上，产生客户交易。后者是我们当前面对的主要问题域用例。我们先把函数定义出来，再思考哪些地方属于死板代码，可以通过宏来生成，让实现更简洁。

代码清单4-14　成交结果分配函数

```
(defn trading?
  "若账户为交易账户，返回true"
  [account]
  (= (:type account) ::trading))

(defn allocate
  "分配成交结果到客户账户并产生客户交易"
  [acc exe]
  (cond
    (nil? acc) (throw (IllegalArgumentException.
                 "账户不可为空"))
    (= (trading? acc) false) (throw (IllegalArgumentException.
                               "必须为交易账户"))
    :else {:ref-no (generate-trade-ref-no)
           :account acc
           :instrument (:instrument exe) :value (:value exe)}))
```

❶ 验证

观察`allocate`函数的内容，它的核心业务逻辑位于`cond`语句的`:else`子句中。前面两个条件子句❶是交易子系统内任何操作都必须满足的验证。对于任何作用于交易的方法，我们都要确认交易账户是非空的实体，还要确认它确实是一个交易账户而非结算账户。以上内容构成了`allocate`方法的主要界面。`allocate`方法含有一些非本质的代码复杂性，有必要分解到核心的API实现之外。

4.5.3 Clojure宏之美

既然我们在allocate方法内做的那些验证其实广泛适用于所有的交易功能，何不把它们重构成可重用的实体呢？那样的话，我们将获得一个可以检查所有账户的验证函数，类似于4.4.1节定义Ruby模块TradeClassMethods时我们对trd_validate所做的处理。不过这次所用的手段——宏——有其鲜明的优点，请看插入内容。



Clojure的宏特性可用于在**编译阶段**生成代码；编译后，宏被展开成普通的Clojure代码成分。优点有两重：

(1) 宏在编译阶段被内联展开，避免了函数调用的开销；

(2) 代码更具可读性，因为不需要用到lambda表达式。如果用高阶函数来实现的话，lambda表达式不可避免。

下面的例子可以让你看清楚宏的用法和特点。例中定义了一个宏，它用在语句里面形式上很像一般的Clojure控制抽象，但其实里面封装了验证逻辑。

```
(defmacro with-account
  [acc & body]
  `(cond
    (nil? ~acc) (throw (IllegalArgumentException.
                        "账户不可为空"))
    (= (trading? ~acc) false) (throw (IllegalArgumentException.
                                      "必须为交易账户"))
    :else ~@body))
```

注意，body内可含有不定数量的form，它们在非符号类型拼接（splicing unquote）运算~@的作用下被插入到生成的代码中。（关于非符号类型拼接的工作原理，详情请参阅4.7节文献[2]。）如果我们把验证逻辑实现成一个函数，不可避免地要用到lambda表达式。而当我们用with-account宏来定义allocate函数时，代码会十分清晰易懂：

```
(defn allocate
 "分配成交结果到客户账户并产生客户交易"
 [acc exe]
 (with-account acc
   {:ref-no (generate-trade-ref-no)
    :account acc
    :instrument (:instrument exe) :value (:value exe)}))
```

现在，实现只需要重点关注核心的领域逻辑，与代码清单4-14相比简洁明了得多。全部的异常处理部分，还有全部的非本质复杂性，完全被分解出来放在宏里面，同时还没有增加任何运行时开销。with-account宏的功用不再局限于allocate的实现；它成了一个通用的控制结构，看上去和一般的Clojure语句成分没什么两样，而且可以被所有需要验证交易账户的API重用。

本节要点

本节的重点是**使DSL实现简洁而又不失表现力**。这个思路也贯穿了全章，各节在阐述过程中呈现的差别只在于如何实现这个思路。

4

Clojure宏除了可使DSL简洁，还**对运行时性能没有任何影响**。Clojure语言本身的可塑性非常强，允许你精确地表达DSL所需的语法和语义。Lisp家族这方面的能力非常突出，天生适合用于建模真实的世界。

不同形式的生成式DSL都可以代替你写代码。但借助敏锐的观察力，你肯定已经察觉不同语言实现，以及不同代码生成时机之间的差别。4.4节讨论了运行时代码生成，本节讨论如何通过Clojure宏实现编译时代码生成。两种策略各有优缺点，你必须仔细掂量所有的选项才好决定一个最佳的实现方案。

4.6　小结

本章漫长的学习之旅已接近尾声，你的耐心值得称赞。我们一路针对金融中介系统领域的问题片段展开讨论，几乎涵盖了所有的内部DSL实现模式。

要点与最佳实践

设计内部DSL的时候，你应遵循实现语言的最佳实践。按照一种语言的习惯去运用它，我们总是能在表现力和性能之间取得最佳的平衡。

Ruby、Groovy等动态语言给予使用者非常大的元编程能力。请借助这些能力去设计DSL抽象和背后的语义模型，你会得到漂亮的简洁语法，而死板代码则交由语言运行时去处理。

对于像Scala这样的实现语言，静态类型是你的好帮手。我们建议大家运用类型抽象来表达大部分业务逻辑，让编译器充当DSL语法的第一道验证防线。

对于Clojure这类具有编译时元编程能力的语言，请用宏来自定义语法结构。你会得到不输于Ruby实现的简洁语法，同时不会遭遇额外的运行时负担。

Ruby、Groovy等动态语言提供了强大的反射式元编程范式，用在DSL实现中对语言简洁性和表现力都有很大的帮助。这类语言允许你在运行时操控其元模型，因此特别适合用来实现较为动态的结构。

本章向你演示了动态builder和装饰器的制作方法，教你驾驭元编程的力量。而且，相信你还学会了运用静态类型以声明式风格表达领域约束。最后，我们学习了如何实现生成式DSL，在编译时或运行时生成代码。

阅读完本章，你应该能够从语言惯用法和最佳实践的角度去思考问题，以之为标杆来衡量自己的DSL实现。我们谈了很多模式，你不难在自己的DSL模型中为它们找到适用的上下文。前面几章一直泛泛地赞扬基于DSL的开发，本章终于把问题域的特定场景和具体的实现结构联系起来了。本章为你的DSL风景线增添了以下主要“景色”：

- 你现在知道如何发挥实现语言内在的力量去提高DSL的简洁度；
- 如果选用静态类型语言，你可以围绕类型化的抽象去建立DSL模型；

- 如果实现语言具备元编程能力，你可以利用这一点使DSL的语法更紧凑，让语言运行时或者编译设施代替你生成代码。

接下来我们应该继续探索更多来自现实世界的例子。下一章我们会讨论动态类型语言家族，看看它们对于实现具有良好表现力的DSL有什么好办法。还等什么呢？让我们精神倍增地学习后面的精彩内容吧！

4.7 参考文献

[1] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[2] Konig, Dierk, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. 2009. *Groovy In Action*, Second Edition. Manning Early Access Program Edition. Manning Publications.

[3] Halloway, Stuart. 2009. *Programming Clojure*. Pragmatic Bookshelf.

[4] Odersky, Martin, Lex Spoon, and Bill Venners. 2008. *Programming in Scala: A Comprehensive Step-By-Step Guide*. Artima.

[5] Coplien, James O. *Design Pattern Definition*. http://hillside.net/patterns/222-design-pattern-definition.

第 5 章

Ruby、Groovy、Clojure语言中的内部DSL设计

本章内容

- 利用鸭子类型和元编程使DSL更简洁
- 用Ruby语言实现交易处理DSL
- 用Groovy语言改进之前的指令处理DSL
- 用Clojure语言转换思路重新实现交易处理DSL
- 一些实现语言的常见陷阱

学习新范式和新设计手段的最佳途径，是找到最能体现该范式特点的语言，观察语言和范式在真实的实现案例中的表现。第4章我们谈了不少有利于提高内部DSL表现力的惯用法和模式。本章我挑选了三种目前最流行的JVM语言，向你示范如何用它们建立符合现实需要的DSL。

本章将按照图5-1所示内容展开讨论。

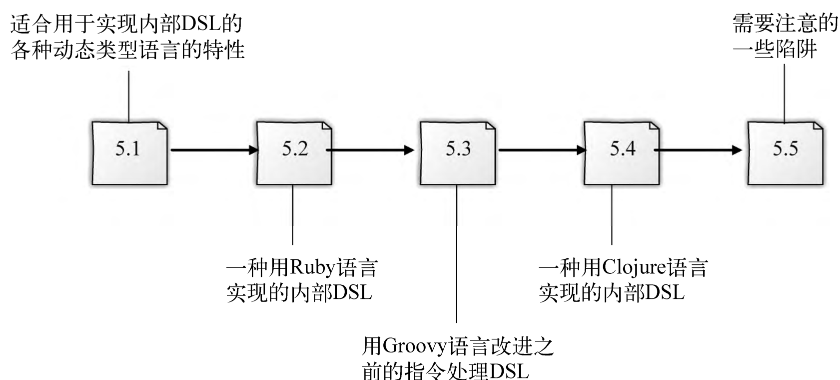


图5-1　本章路线图

本章选择的语言都是动态类型语言。我首先会说明动态类型语言拥有哪些特质能造就优秀的DSL，其次解释为何选择Ruby、Groovy、Clojure这三种语言来展开讨论。然后我们进入实现环节，

依次引导你用这三种语言分别实现一个完整的DSL。这期间我们会讨论每种语言中常用于DSL设计的主要特性，同时介绍选取正确实现模式的若干原则和思路。

阅读完本章，你对于如何在具有类似特性的语言中开展DSL设计工作将有一个全面的认识。经过多个完整的DSL实现过程，你将学会从目标DSL的角度去思考，学会按照设计意图把实现语言编排成你希望提供给用户的DSL语法。

本章涉及大量编程实践。请你把相应语言的解释器都找出来，准备好迎接大量代码。我们的例子都比较简短明了，保证不会让你厌烦。本书附录为三种语言都准备了简单的复习资料，如果你对哪种语言不太熟悉，不妨翻看一下作为热身。如果你对多语开发，即同时使用多种语言进行开发的概念感到陌生，附录G可以作为入门介绍。下面我们就从选择这三种语言的理由说起。

5.1 动态类型成就简洁的 DSL

内部DSL将领域语义呈现为更易读的形式，这是内部DSL在实现语言上面增加的一种重要性质。内部DSL用领域用户能明白的语句、语汇向用户解释实现的含义。



当没有程序员背景的领域专家阅读一段DSL脚本时，他应该能够读懂其中的领域规则。能够疏通开发者与领域从业者之间的沟通渠道，就是DSL的真正价值。我不鼓吹让每个非程序员背景的领域专家都能用DSL**编写**程序，但至少应该做到让领域专家能够**理解**DSL脚本中的领域语义。

5

动态类型语言写成的程序不带类型标注，这本身就减少了视觉上的干扰，可以更清楚地说明编程者的意图。这样写出来的代码可读性较好，而易读性正是DSL区别于一般API的基本特质之一。我会用下面几节说明用动态类型语言开发出来的DSL所具备的三大重要特质：

- 更易读，因为没有类型标注的干扰（5.1.1节）；
- 鸭子类型，涉及DSL接口契约的设计思路（5.1.2节）；
- 元编程，从DSL实现中清除死板代码的一种方式（5.1.3节）。

5.1.1 易读

DSL的读者都希望语言自然流畅，不愿意其中掺入不必要的复杂内容。编程语言的类型系统有可能助长DSL的非本质复杂性。如果你用了一种类型系统像Java那么琐细的实现语言，很可能最后出来的DSL要在抽象身上附加一串不必要的类型标注。动态类型语言不要求提供类型标注，所以相比同等情况下的静态类型语言实现，能更清楚地表达编程者的意图。至于意图背后的实现，倒不一定更容易理解。（5.5节将讨论基于动态语言的DSL实现中的常见陷阱，届时你会看到这方面的例子。）总体而言，动态类型语言的语法较为简明，制作出来的DSL及其实现也因此较为易读。

DSL是否易读可以直观地感受出来，不过动态语言还有另一个特点，同样在内部DSL的设计和实现中扮演了重要角色。动态类型语言轻仪节而重语义。它们比静态类型语言言辞精炼，虽然

表现出来的还是抽象组成的层次结构，但背后的思维是不同的。不同的地方在于抽象如何响应发送给它的消息。

5.1.2　鸭子类型

鸭子类型不一定是弱类型。假设你在一个对象上调用某消息，如果该对象满足消息所要求的契约，就会得到响应；否则，消息将沿着该对象的继承链向上传播，直至找到一个祖先对象满足消息契约为止。如果一直上溯到根对象都找不到合适的方法来响应该消息，用户将得到一个`NoMethodError`。编译时并不确定在这个对象上调用那则消息是否有效，因为不存在这方面的静态检查。但用户可以在运行时修改对象的响应目标，改变其方法和属性。对于任意给定的消息，如果在消息被调用的那一刻，目标对象支持该消息，那么就认为调用是有效的。这个机制称为鸭子类型（duck typing），Ruby、Groovy、Clojure等众多语言都实现了这种机制。静态类型语言如Scala，也可以实现鸭子类型，我们将在第6章论及。

1. 通过鸭子类型实现的多态

动态语言中的鸭子类型对于我们实现DSL有什么好处呢？简单来说还是那句话，我们牺牲静态类型安全换取简洁的实现。你不需要静态地声明接口，也不需要依赖继承关系去实现多态。只要消息的接收者实现了正确的契约，它就可以合理地响应消息。图5-2描绘了通过鸭子类型来实现多态的情形。

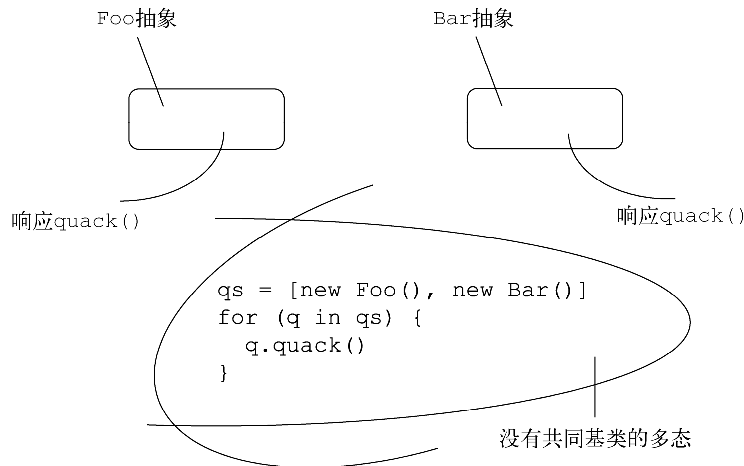


图5-2　鸭子类型构成的多态。Foo和Bar抽象并没有任何共同的基类，但在支持鸭子类型的语言里，可以把它们看作是多态的

接下来我们看一个金融交易领域的例子。我们首先做一个用了接口的Java实现，然后做一个用了鸭子类型的Ruby实现，让你在对比之下体会后者的简洁。

2. 金融交易领域的例子

交易所里履行的交易有许多类型，类型的划分与被交易的票据种类有关。（交易、票据、成

交这些概念你应该已经熟悉了，万一有所遗忘，请翻查前面几章的插入栏。）证券交易是涉及股票和固定收益的买卖。外汇交易涉及以即期（spot）或掉期（swap）交易的形式交换各种外国货币。一般地，在Java这样的静态类型语言里面，你会将这两个概念建模为同一接口下的两个特殊化抽象。继承链的定义也是静态的，如下面的代码片段所示：

```
interface Trade {
  float valueOf();
}

class SecurityTrade implements Trade {
  public float valueOf() { //..  }
}

class ForexTrade implements Trade {
  public float valueOf() { //.. }
}
```

如果有一个计算交易现金价值的方法，要求可以传递给它任意类型的Trade对象，那么实现出来会是这个样子：

```
public float cashValue(Trade trade) {
  trade.valueOf();
}
```

cashValue方法的参数被限定为实现了valueOf方法的最高一级静态类型。这项约束由Java编译器静态地核查。作为对比，请你看另一个实现，下面的代码清单用了鸭子类型来实现cash_value方法。

5

代码清单5-1 鸭子类型构成的多态

```
class SecurityTrade
  ## ..
  def value_of
    ## ..
  end
end

class ForexTrade
  ## ..
  def value_of
    ## ..
  end
end

def cash_value(trade)
  trade.value_of
end
cash_value(SecurityTrade.new)
cash_value(ForexTrade.new)
```

这个实现没有前面那些周边设施，不存在静态继承关系；只要传递的对象实现了value_of方法， cash_value就能执行无误。假如传递一个没有实现value_of方法的无关对象，会怎么样呢？毫无疑问，cash_value会在运行时卡壳。所以你应该有全面的单元测试去核查保护各处契约。

单元测试时，由于不需要穷于应付静态类型的安全，很容易构建用于测试的模拟对象。记住，在一种支持鸭子类型的语言里面，不要去检查类型，也不要试图用动态语言来模拟静态类型。这是一种截然不同的抽象设计思路。抽象有没有实现它应该向客户提供的契约，这才是你的测试套件所要针对的目标。

鸭子类型令你无需写出静态的约束检查代码。仅仅这一条就立即让DSL实现简洁了许多，同时编写者的意图也表达得很清晰。我们在4.2.2节，在Ruby中通过动态mixin实现装饰器的时候已经讨论过这一点。最后得到的DSL如下所示：

```
Trade.new('r-123', 'a-123', 'i-123', 20000).with TaxFee, Commission
```

注意，我们向`Trade`抽象里混入了`TaxFee`和`Commission`两个模块，交易的总现金价值通过Ruby的鸭子类型计算得出。

接着我们要与第4章认识的元编程技术再次碰面。动态类型语言用元编程技术可以把你从重复性的死板代码中解放出来。

5.1.3　元编程——又碰面了

除了省去类型标注外，动态类型还有什么办法令DSL语言更简洁？答案我们在上一章就知道了，它可以通过语言本身的机制生成重复性的代码结构，免除手工编写的负担。简洁的实现对于DSL的设计者很重要，而简洁的DSL编程界面对于DSL的使用者同样重要。Ruby和Groovy都拥有非常精良的元编程设施，可以在运行时引入新的方法和属性，我们已经在2.3.1节和4.2节讨论过。举一个例子来回顾动态类型对于DSL实现简洁性的正面影响吧。下面的代码清单用Groovy语言演示了一个运用元编程和闭包来实现的XML建造器。

代码清单5-2　Groovy语言实现的XML建造器，展示动态元编程的力量

```
def clientOrders = //..
builder = new groovy.xml.MarkupBuilder()

builder.orders {
  clientOrders.each {ord ->
    order(type: ord.getBuySell()) {          <-- ❶ Groovy的动态方法分发
      instrument(ord.getSecurity())
      quantity(ord.getQuantity())
      price(ord.getLimitPrice())
    }
  }
}
```

例中Groovy语言实现的`MarkupBuilder`对于`order`、`instrument`、`quantity`、`price`等方法一无所知❶。语言运行时通过Groovy的动态方法分发机制以及`methodMissing()`钩子，拦截所有未定义的方法调用。Ruby语言也有类似的手法。动态类型语言提供了拦截器来应付所有未定义的方法。这样的技巧使程序更简洁、更动态，同时又能保留必要的表现力。

我们分析了动态语言实现的DSL的三项代表性特质。第一项易读性说的是DSL脚本的表面语法；另外两项，鸭子类型和元编程则更多地与实现技术有关。我们试着列举一下Ruby、Groovy、

Clojure语言各具备哪些特性，有利于创作表现力充沛的DSL。

5.1.4 为何选择Ruby、Groovy、Clojure

Ruby、Groovy、Clojure三种语言都完全具备前述动态类型语言的三大特质，它们都是适合实现内部DSL的优秀宿主语言。表5-1总结了它们的语言特性。

表5-1 Ruby、Groovy和Clojure语言具备以下特质，使它们成为内部DSL实现语言的优秀候选者

	易 读 性	鸭子类型	元 编 程
Ruby	语法灵活，无需类型标注，文字表达手段丰富	支持鸭子类型，且可用`responds_to?`检查一个类是否响应给定的消息	具有很强的反射式和生成式元编程能力
Groovy	语法灵活，类型标注可选，文字表达手段丰富	支持鸭子类型；允许无公共基类的多态	运行时元编程能力强，通过Groovy元对象协议（MOP）实现
Clojure	语法灵活，但作为Lisp的变体，为前缀语法形式所限。允许程序员提供可选的类型提示（type hint）以利方法分发，可避免像Java那样的反射式调用	如同Ruby或Groovy，也支持鸭子类型	可经由宏机制实现编译时元编程。Clojure拥有极强的可塑性，可视DSL之需灵活扩展

虽然Ruby、Groovy、Clojure有一些共同的特点，但它们在DSL实现方面的差别其实也很大，足以使我们分成三节来分别讨论。这三种语言都在JVM上运行，都拥有很强的元编程能力，也都迅速成为了主流的开发语言。然而就在与JVM集成的方式这个很基本的问题上，它们却给出了不一样的答案。图5-3总结了这三种语言的一些异同。

	JRuby	Groovy	Clojure
运行在JVM上	X	X	X
动态类型	X	X	X
运行时元编程	X	X	
编译时元编程	?	?	X
共享Java的对象系统		X	X
桥接Java的对象系统	X		

？：通过操纵AST的库提供有限支持

图5-3 Ruby、Groovy、Clojure呈现了多样化的DSL实现方案

本章我们将分别用这三种语言探索内部DSL的实现。在讨论过程中，我们将观察每一种语言的不同特性，同时剖析每项特性在第4章深入讨论过的那些模式中所起的作用。

5.2 Ruby 语言实现的交易处理 DSL

本节我们要开发一个完整的用例。我们要设计一种DSL用于建立新的证券交易，并根据可灵活组合的业务规则计算交易的现金价值。DSL执行之后，将产生一个`Trade`抽象的实例供应用程序使用，具体用法因应用而异。我们会从一个简单的实现开始并逐步进行改进，充实其表现力和领域内涵。图5-4展示了DSL演进的迭代路线图。

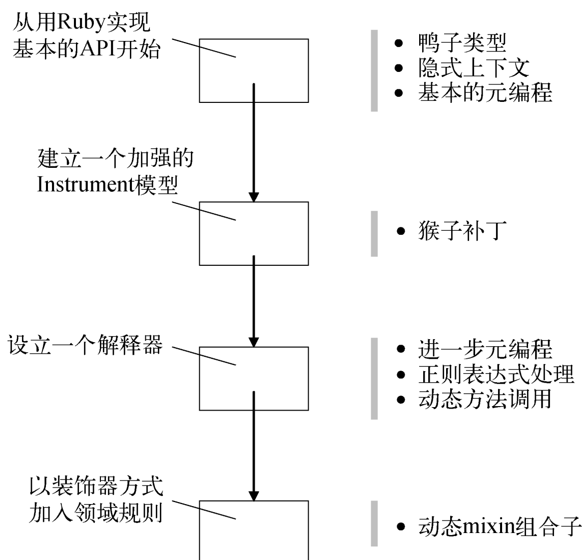


图5-4　我们逐步丰富Ruby DSL的实现手段，完善交易处理DSL。每个阶段我们都投入更多Ruby语言的抽象能力，增加更多领域功能，用以充实DSL

在整个开发过程中，老鲍会担当我们的指导，负责指出所有不当之处和需要改进的地方，帮助我们打造一个表现力充沛的DSL设计。至于能不能满足老鲍的要求，就要看Ruby帮不帮忙了。

代码提示　后面几节含有大量代码片段，我会插入说明一些预备知识，解释必要的语言特性，以便读者理解实现中的细微之处。阅读之前也不妨先翻看本书附录中相应语言的速查表格。

请牢记我们的目标：DSL要让老鲍能看懂，并且能检查DSL是否违反了他的业务规则。

5.2.1 从API开始

最开始的API设计总是略显粗糙的。动态语言的开发历程就像制陶一样，你总是从一团黏土开始，然后有步骤地塑造其形态，逐渐增加其表现力。


Ruby知识点

- **Ruby中如何定义类和对象？** Ruby是一种面向对象语言，类的定义形式与其他OO语言相仿。Ruby有其特殊的对象模型，允许你在运行时通过元编程机制修改、调查、扩展对象。
- **如何使用散列容器实现不定长的参数列表？** Ruby语言允许你向方法传递一个散列容器作为参数，以此来模拟“关键字参数”（keyword arguments）的特性。
- **Ruby元编程基础知识。** Ruby的对象模型包含许多可以用于反射式和生成式元编程的元件材料，例如类、对象、实例方法、类方法、单例（singleton）方法，等等。Ruby元编程机制允许你在运行时探查其对象模型，也允许动态地改变行为或生成代码。

请思考以下代码片段，这是我们的API设计师想出来的第一版DSL：

```
instrument = Instrument.new('Google')        <-- 将被交易的新票据
instrument.quantity = 100

TradeDSL.new.new_trade 'T-12435',            <-- 将被创建的交易
  'acc-123', :buy, instrument,
  'unitprice' => 200,
  'principal' => 120000, 'tax' => 5000
```

老鲍见了之后大声嚷嚷起来：“喂！这个东西太技术了。我想要一个票据对象还得调用那一串奇怪的构造？我平常可不是这么解读交易和票据的。”

老鲍的话有道理，我等一下再解释。你先跟我复诵一遍：DSL绝不会第一次就做对。DSL总是迭代演进的。上面的片段只是一套中规中矩的API，其易读性只达到Ruby的一般水平。它还没有形成连贯的句子，读起来不像老鲍平常处理交易业务时挂在口边的话语。不过，这段代码给我们奠定了一点基础，可以作为我们的出发点。

1. 基本抽象

任何DSL设计都是从一组基本抽象开始，然后在上面建立符合领域习惯的语言。这样的过程我们称为自底向上的编程方式，大的抽象由小的抽象生长而成，最后形成领域专家想要的表现形态。

我们的DSL设计从针对`SecurityTrade`、`Instrument`等基本领域实体的一组API开始。下面的Ruby代码清单是API对应的一些基本抽象。

代码清单5-3 `SecurityTrade`的Ruby实现（第一次迭代）

```
class SecurityTrade

  attr_reader :ref_no,
              :account,
              :buy_sell,
              :instrument,
```

```
              :unitprice

  attr_accessor :principal,
                :tax,
                :commission

  def initialize(ref_no, account, buy_sell, instrument, unitprice)
    @ref_no = ref_no
    @account, @buy_sell, @instrument, @unitprice =
      account, buy_sell, instrument, unitprice
  end

  def self.create(ref_no, account, buy_sell, instrument, h)      ❶ 用于创建交易的类方法
    tr = new(ref_no, account, buy_sell, instrument, h['unitprice'])
    [:principal, :tax, :commission].each do |m|
      tr.instance_eval("tr.#{m} = h['#{m}'] if h.has_key?('#{m}')")
    end
    tr
  end                                                            填入来自散列容器的值 ❷
end
```

create类方法里面的h是个散列容器❶，用来给unitprice、principal和tax提供命名参数。用散列容器来实现命名参数是Ruby的一种惯用法。位置❷还采用了一种有意思的技巧，即利用元编程建立被调用对象的隐式上下文，并用散列容器h中取出的各参数值填充交易对象实例。我们在4.2.1节讨论过如何建立隐式上下文。

代码清单5-4实现了Instrument类。这个类没有写成不可变的形式，除此之外没什么特别值得注意的地方。就当前版本的DSL而言，我们可以把它写成不可变的值对象。但之所以保留为可变的对象，到下一节你就能清楚知道其理由，到时候我们要利用它的可变性来创造一种票据DSL。

代码清单5-4　在Ruby中实现Instrument交易

```
class Instrument
  attr_accessor :name, :quantity
  def initialize(name)
    @name = name
  end

  def to_s()
    "(Name: " + @name.to_s           +
    "/Quantity: " + @quantity.to_s   + ")"
  end
end
```

本节代码缺少的最后模块是TradeDSL类，它只负责把前面的材料串在一起：

```
require 'security_trade'
class TradeDSL
  def new_trade(ref_no, account, buy_sell, instrument, attributes)
    SecurityTrade.create(ref_no, account, buy_sell, instrument, attributes)
  end
end
```

我们的DSL迈出了第一步。你会在后续的迭代中观察到enter code hereTradeDSL的成长过程，它的表现力会越来越强，我们也会逐步加入更多的功能。

2. DSL门面

TradeDSL类演示了一种让DSL语法与底层实现解耦的重要手法。一方面，这个类向用户呈现DSL表面语法；另一方面，它把基本抽象包装起来，在实现之上插入一个间接层。图5-5形象地描绘了DSL的这种结构。

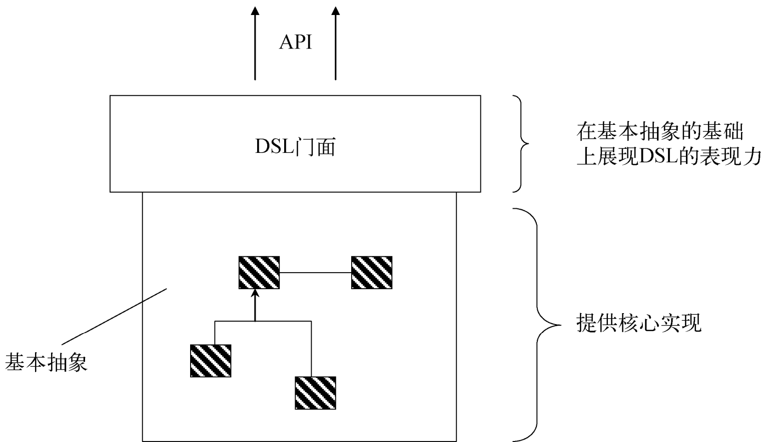


图5-5 DSL门面既向用户提供表现力充沛的API，又保护核心实现结构不被暴露

切记，在设计DSL时，一定只向用户提供单一的交互点。本例中TradeDSL类担当了DSL门面角色。目前这个门面仅包装了SecurityTrade类的create方法，我们将在后续的迭代中持续地充实、完善这个抽象，使它能够满足用户的需求。现在最紧迫的任务是解决老鲍对DSL中创建票据部分的不满。这种时候来点不按常规的办法反而管用。

5.2.2 来点猴子补丁

TradeDSL类的下一步进化目标是让老鲍更轻松地创建票据。他想按照平常在交易台前的习惯那样，购买100股IBM的股票。下面是一段创建交易的DSL，里面说明了被交易的票据详情，我们就希望能把脚本写成这个样子。

```
TradeDSL.new.new_trade 'T-12435',
  'acc-123', :buy, 100.shares.of('IBM'),
  'unitprice' => 200, 'principal' => 120000, 'tax' => 5000
```

之前那些老鲍看不懂的多余句法结构不见了，他可以用平常习惯的语言来设立票据：100.shares.of('IBM')。老鲍很满意！那么我们费了哪些功夫才得到这样的脚本呢？

> **Ruby知识点**
>
> **猴子补丁**（monkey patching）指在已有的类中加入新的属性或方法。Ruby允许打开一个已经存在的类，引入新的方法和属性来扩增类的行为。这项特性极为强大，强大到你可能会忍不住滥用它。

代码清单5-5实现了shares和of两个方法，我们不动声色地把它们引入到Numeric类。Numeric是Ruby语言内建的类，任何Ruby类都可以被打开并引入新的属性和方法。这样的做法称为猴子补丁（monkey patching）。很多批评者认为不应该鼓励使用这种特性，因为猴子补丁威力太大，使用的时候不得不注意其风险和陷阱。任何一本正经的Ruby教科书（5.7节文献[1]）都会警告你不要过度使用。其实，只要谨慎使用，猴子补丁可以给DSL插上翅膀。

代码清单5-5　使用了猴子补丁的票据DSL

```
require 'instrument'
class Numeric                                  ❶ 打开Numeric类
  def shares                                   ❷ 新方法shares
    self
  end

  alias :share :shares

  def of instrument                            ❸ 新方法of
    if instrument.kind_of? String
      instrument = Instrument.new(instrument)
    end
    instrument.quantity = self
    instrument
  end
end
```

写完这段代码，交易DSL的第一次迭代就告一段落。随着核心抽象的各部分逐渐延伸融合，我们DSL的表现力也越来越强。5.2.1节开头那段代码在创建票据时伴随的干扰现在已经被清理掉。不过与老鲍想要的自然语言表达相比，我们的实现还存在不少句法怪异的地方。Ruby有办法帮我们更进一步，而且我们的TradeDSL门面也经得起折腾。下一节我们会用一点语法糖衣将它装点起来，打扮成老鲍满意的DSL。

5.2.3　设立DSL解释器

表现力要多强才算足够？这个问题没有固定的答案，因DSL用户的立场而异。对于熟悉Ruby的程序员来说，第一次迭代的DSL表现力已经足够。即使是不懂编程的领域专家，也大体知道写的是什么，只不过有些多出来的句法结构看起来可能不太舒服而已。因为精益求精，也因为Ruby语言有这样的能力，所以我们可以把DSL做得更完美一些，更接近老鲍在交易台前所说的语言。



Ruby知识点

- **如何利用Ruby的“嵌入文档”（here documents）特性定义多行字符串**。通过这个技巧，可在源代码中直接定义一段文本，而不必另行写到代码外部。
- **如何定义类方法。类方法**（或单例方法）是Ruby单例类（singleton class）的实例方法。详情请查阅5.7节文献[1]。
- **evals的一般用法及其元编程用途**。在运行时动态地求解一串或一段内含代码的字符，是Ruby最强大的特性之一。Ruby的evals有好几种形态，适用于不同的上下文。

- **Ruby正则表达式的处理**。Ruby内置支持正则表达式，对模式匹配和文本处理有极大的帮助。

1. 加入解释器

我们在5.2.2节已经为TradeDSL开发了相当有表现力的语法，能出色地反映领域语义。不过对老鲍来说，还是技术味太浓了一点，他习惯于说更流畅的交易语言。

第二次迭代我们准备推出一个解释器来翻译老鲍的语言，把他的话去芜存菁，提取构建抽象所需的必要信息。本次迭代完成之后，DSL脚本看起来应该是这个样子：

```
str = <<END_OF_STRING
  new_trade 'T-12435' for account 'acc-123'
                      to buy 100 shares of 'IBM',
                      at UnitPrice=100, Principal=12000, Tax=500
END_OF_STRING

puts TradeDSL.trade str
```

DSL的核心抽象上一次迭代时就已经准备就绪，我们现在开始动手制作语法糖衣。我们的交易处理语言正按照计划稳步前进。

要让代码变成上面的样子，我们需要往TradeDSL类里加些什么东西呢？5.2.2节打造的门面TradeDSL，经过第二次迭代变成代码清单5-6的样子。类中设立了一个小小的解释器，负责将用户输入处理之后再传递给SecurityTrade。

5

代码清单5-6 在Ruby中将交易DSL做成解释器的样子（第二次迭代）

```
require 'security_trade'
require 'numeric'

class TradeDSL
  class << self
    def const_missing(sym)                    ❶ 拦截未定义的常量
      sym.to_s.downcase
    end
    def trade(str)
      TradeDSL.new.interpret(str)
    end
  end

  def new_trade(ref_no, account, buy_sell, instrument, attributes)
    SecurityTrade.create(ref_no, account, buy_sell, instrument, attributes)
  end

  def interpret(input)                        ❷ 提供隐式上下文给new_trade
    instance_eval parse(input)
  end
                                              ❸ 处理用户输入
  def parse(dsl_string)
    dsl = dsl_string.clone
    dsl.gsub!(/=/, '=>')
    dsl.sub!(/and /, '')
    dsl.sub!(/at /, '')
    dsl.sub!(/for account /, ',')
```

```
    dsl.sub!(/to buy /, ', :buy, ')
    dsl.sub!(/(\d+) shares of ('.*?')/, '\1.shares.of(\2)')
    dsl.sub!(/(\d+) share of ('.*?')/, '\1.shares.of(\2)')
    puts dsl
    dsl
  end
end
```

与其直接解释每一行代码做了什么，不如先看一幅示意图。图5-6描绘了老鲍的语言被解释的全部步骤。

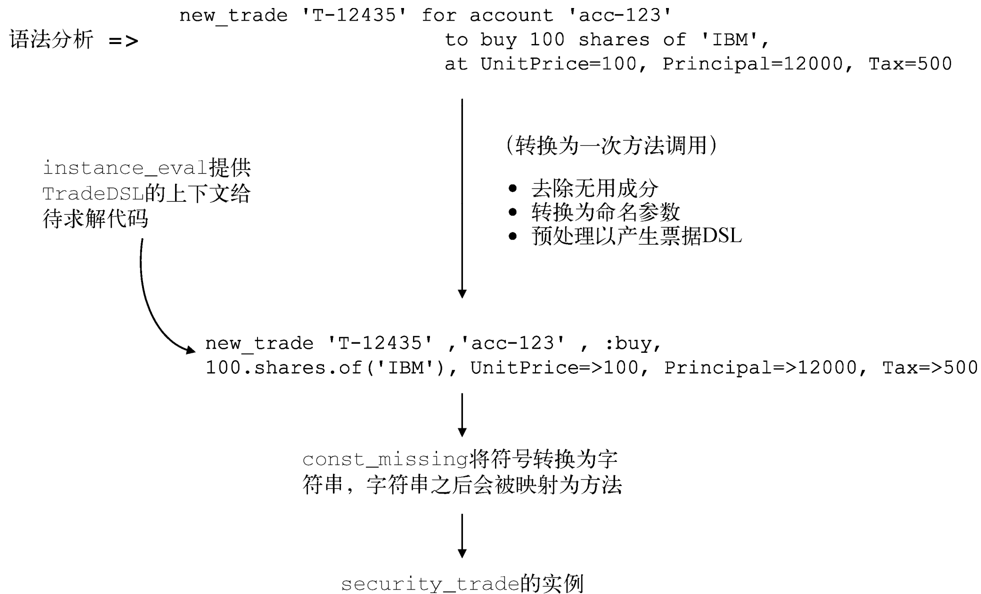


图5-6　一段TradeDSL脚本示例被代码清单5-6的程序解释并生成Ruby对象的全过程。经由DSL解释器生成了一个security_trade实例

请把这幅图与代码清单5-6对照着来理解。你能发现其中用了第4章介绍过的哪些手法吗？还真不少，三不五时就改头换面地冒出来一个。下面的列表可以帮你发现几个：

- const_missing方法❶用了运行时元编程（见4.4节）手法，它将任何未定义的常量转换为字符串；
- interpret方法中的instance_eval❷将TradeDSL的一个实例设立为执行new_trade方法的隐式上下文（见4.2.1节）；
- parse方法使用正则表达式❸处理用户输入，将输入转换为符合new_trade方法调用形式的字符串。

若想更深入了解Ruby元编程技术，请参阅5.7节文献[5]。

2. 像老鲍那样说话

现在从DSL用户的角度回顾一遍前面的工作。用户现在可以用他日常业务活动中的语言写出

生成新交易的DSL脚本。我们在DSL中插入了一些无意义的词汇，以尽量符合一般领域用户的用语习惯。作为用户，老鲍可以把DSL文本写到一个文件内，文本经过加载、处理之后生成`SecurityTrade`的实例。即使交易数据来自上游的营业系统，他照样可以用这DSL生成`SecurityTrade`实例并存入数据库。

下一节我们继续改进DSL，并纳入一些业务规则，一方面便利于程序员用户，另一方面其他的领域用户也可在老鲍生成的交易上增补规则，用于后续的交易环节。

5.2.4 以装饰器的形式添加领域规则

虽然老鲍满意目前的交易生成方式，但他对后续的交易环节仍有些担心，因为后面要在交易上补充业务规则。我们答应老鲍立即着手这个问题，一旦这部分语言的表现力过关就拿给他看。那么我们现在就开始新的迭代，目标是增加DSL功能并充实交易的业务规则。



Ruby知识点

- **如何定义、使用Ruby块**。Ruby语言的块（block）被用于实现lambda和闭包。
- **如何通过模块实现mixin**。Ruby模块（module）是一种组织代码制品的方式。类可以通过mixin的形式包含模块及其中的制品。
- **如何串联不同的mixin**来设计装饰器。
- **鸭子类型**。在Ruby语言中，只要对象实现了与某消息同名的方法，就可以响应该消息。对象是否实现了特定方法不作静态检查；可在运行时修改对象。只要能学鸭子叫，Ruby就认为那是一只鸭子。

1. 交易DSL现状回顾

我们不能一味埋头改进，在着手实现交易充实功能之前，不妨先回顾一下现在所处的位置。图5-7一目了然地展现了目前的进展和后续的任务。我们开发的交易生成DSL会产生一个`SecurityTrade`实例，下一步要把业务规则充实到交易中，可考虑将业务规则建模为DSL。

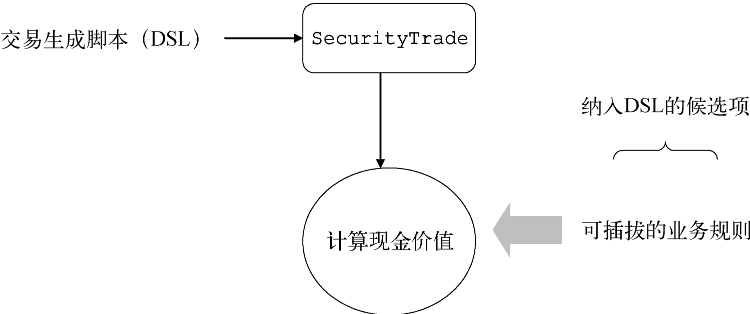


图5-7　我们已经开发了生成交易的DSL，现在要增加用来计算交易现金价值的业务规则。我们准备把业务规则也做成DSL

当交易被送到证券交易机构的后台时，将被附上其现金价值和静态数据资料，为进入处理流程的后续环节做好准备。4.2.2节已介绍过如何计算交易的现金价值，也叫做净结算价值。后台接收到交易之后，需要援用领域规则来计算交易的现金价值。领域规则因交易市场、票据类型等各种因素而异。为免讨论过于复杂，我们假设有一组固定的规则。为了在生成的交易上实施这组规则，我们需要扩展现有的DSL。

2. 实现领域规则

以下规则适用于老鲍生成的交易：

- 交易的现金价值由基本价值总额、税费总额、佣金总额计算得出。
- 如果输入的交易流含有以上总额中的任意项，该项将按输入的数额计算；否则按照以下规则对每笔交易分别计算后汇总得出。
- 以下业务规则适用于所有交易：
  - 基本价值总额为单价与数量的乘积，单价、数量都是交易对象的一部分。
  - 税费按基本价值总额的固定比例计算。
  - 佣金按基本价值总额的固定比例计算。

这几条规则实现之后，用户使用DSL充实交易的情形如下所示。

代码清单5-7　交易DSL的用法

```
require 'trade_dsl'
require 'cash_value_calculator'
require 'tax_fee'
require 'broker_commission'

str = <<END_OF_STRING
new_trade 'T-12435' for account 'acc-123'
                    to buy 100 shares of 'IBM',
                    at UnitPrice = 100
END_OF_STRING

TradeDSL.trade str do |t|                                  ❶ 为了副作用而使用Ruby块

  CashValueCalculator.new(t).with TaxFee, BrokerCommission do |cv|      ❷ 以mixin手法实现的装饰器

    t.cash_value = cv.value                                ❸ 块内发生的副作用
    t.principal = cv.p
    t.tax = cv.t
    t.commission = cv.c
  end
  t
end
```

`TradeDSL.trade(str)`生成的`SecurityTrade`实例被传递到一个Ruby块❶，然后作为`SecurityTrade`实例在块内被修改❸的副作用，完成了交易充实过程。以上写法是常规的、合乎语言习惯的Ruby编程方法。这段脚本的表现力是简洁的Ruby语言和我们加入的领域语义共同作用的结果。

上面的代码将`TaxFee`和`BrokerCommission`的计算逻辑单独抽象出来，做成可插拔的DSL

部件，这一点也值得注意。用户需要什么部件，就把什么部件连接到CashValueCalculator类❷。这样的设计手法就是我们在4.2.2节讨论过的，所谓的“基于mixin的编程”。一个个mixin组件就是附着在主体类CashValueCalculator上的装饰器。

我们还要对TradeDSL.trade方法做一点改动才能让它接受一个块作为参数。DSL的其他部分不变。

代码清单5-8 交易DSL的Ruby实现：为了副作用而使用Ruby块（第三次迭代）

```
require 'security_trade'
require 'numeric'

class TradeDSL
  class << self
    def const_missing(sym)
      sym.to_s.downcase
    end
    def trade(str)
      yield TradeDSL.new.interpret(str) if block_given?    ❶ 处理块，得到副作用
    end
  end

end
```

代码清单5-7还留下了一点未解决的地方，CashValueCalculator实例上很明显的附加了各种装饰器，但我们还没搞清楚它们的实现。

3. 附加装饰器的Ruby DSL

代码清单5-7向你演示了怎样在核心抽象上装点像TaxFee和BrokerCommission那样的语法糖衣。而且与静态语言不同，我们的装点工作可以借助元编程的力量在运行时巧妙地完成。下面的代码清单完整地实现了对给定的交易计算其现金价值的DSL。

代码清单5-9 计算交易的现金价值

```
class CashValueCalculator
  attr_reader :trade

  attr_accessor :p, :t, :c                          ❶ 简洁、含义清晰的句法

  def initialize(trade)
    @trade = trade
    @p = [@trade.principal,
          @trade.unitprice * @trade.instrument.quantity].find do |m|
      not m.nil?
    end
    @t = @trade.tax unless @trade.tax.nil?
    @c = @trade.commission unless @trade.commission.nil?
  end

  def with(*args)
    args.inject(self) { |acc, val| acc.extend val }    ❷ 动态地合成各mixin
    yield self if block_given?
  end

  def value
```

```
    @p
  end
end

module TaxFee
  def value
    @t = @p * 0.2 if @t.nil?
    super + @t
  end
end

module BrokerCommission
  def value
    @c = @p * 0.1 if @c.nil?
    super + @c
  end
end
```

成了！现在我们的DSL外有老鲍能理解的自然语法，内有领域语言表述的清晰实现。5.1节所述的动态类型语言三大特质都体现在我们的Ruby DSL上，表5-2对此作了简要总结。

表5-2　动态语言和Ruby DSL

特　　质	代码清单5-9中用到的Ruby特性
易读性	灵活柔顺的语法、数组字面量、可选的圆括号，这几项特性使`initialize`方法❶内的代码清晰简明。领域规则被直白地表达出来，易于读者理解，如果输入交易时一并输入了现金价值的构成要素，优先按照输入值计算，否则从交易中算出
	交易的现金价值总额由混入到`CashValueCalculator`实例的各模块隐含地计算得出。代码清单5-7的DSL脚本很好地抽象了现金净值的具体计算过程，同时又清楚地告诉用户计算中涉及哪些构成要素。实际上，用户希望最后的净值算入哪些要素，就提供哪些要素
鸭子类型	注意`TaxFee`和`BrokerCommission`的`value`方法都是在没有任何静态继承关系的情况下使用了super关键字。这是对鸭子类型的一次应用
	只要插入任何一个拥有`value`方法的模块，都能顺利完成计算
元编程	`with`方法❷起到了组合子的作用，通过在运行时扩展各参与模块，实现对各mixin的组合

交易DSL的Ruby实现至此全部完成。我在本节开头提出一个待解决的现实用例，然后向你演示基于DSL的问题解决方式。现在我们看到了结果，并为打算建模的领域功能找到了最自然的实现方式。借助Ruby语言的灵活语法、鸭子类型和元编程能力，最终创造出一种领域专家能完全领会的专用语言。我们一边改进实现，一边着重学习了那些使Ruby成为优秀内部DSL实现语言的特性。这样安排不是为了卖弄Ruby的能力，而是反复向你演示在基于DSL的开发方式下，如何配合强力的编程语言去创造具有扩展性的抽象。

下一节将探讨另一种DSL实现语言，它像Ruby一样具有动态类型和强大的元编程能力，而且它可以和JVM更紧密地集成。我们在第2章和第3章设计指令处理DSL的时候已经用过它——Groovy语言。现在我们准备再对之前的实现做一些改进。



你有没有想过既然平常的开发多半只会用一种语言，那么我们为什么要学习那么多种语言呢？在现实的开发中，最切合解答域需要的语言才是最理想的DSL实现语言。请记住DSL的语法和语义才是最重要的决定因素；选择哪种实现语言只是手段而已。你学到手的套路越多，设计DSL的时候能使的招式就越多。

5.3　指令处理 DSL：精益求精的 Groovy 实现

以语言的能力来说，Groovy与Ruby较为接近，都支持鸭子类型，也都有很强的运行时元编程的能力。两种语言的主要区别在于Groovy共享了Java的对象模型，因此Groovy的无缝集成能力比Ruby强。实际上，Groovy本身就常被作为Java语言的一种DSL来宣传。因此，如果你的DSL需要融入Java应用的大环境，Groovy是一种合适的实现语言。作为DSL的宿主语言，Ruby和Groovy的实现能力相近。但由于Groovy能共享Java的对象系统，它的集成能力更强一些。

本节我们再次翻新之前在第2章和第3章先后实现、加强过的指令处理DSL。Groovy有一些与Ruby相似的特性，我们上一节用Ruby实现交易DSL时已经讨论过，本节不再着重介绍。本节我们的讨论重点是Groovy一项特别突出的元编程特性，你在设计内部DSL的时候会常常用到它。

在正式展开讨论之前，我们将简要回顾指令处理DSL的前几次迭代，分析其中的不足，然后我们持续改进，直至最后版本的完善实现。

5.3.1　指令处理DSL的现状

我们已经讨论过多种Groovy实现选项，简要总结如图5-8所示。

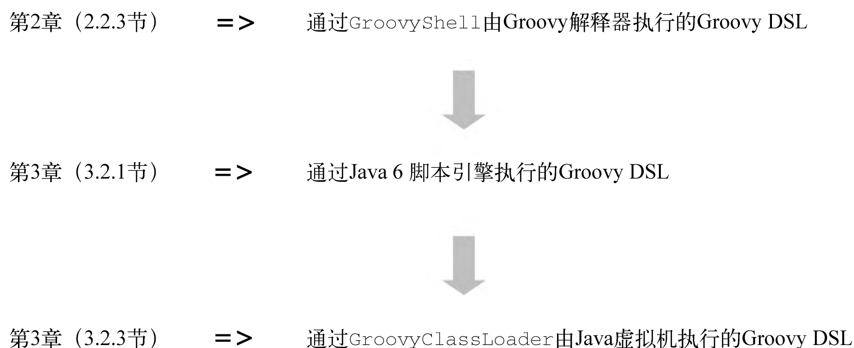


图5-8　前面章节中尝试过的多种指令处理DSL实现方案

通过`GroovyShell`在Groovy环境内执行DSL，我们在2.2.3节从头到尾完成过一个Groovy实现。`GroovyShell`用它的`evaluate`方法执行接收到的DSL定义和脚本。在3.2.1节，我们修改了DSL，并改用Java 6脚本引擎API来执行。最后在3.2.3节，我们用更好的选择取代了3.2.1节的方案，不再假手脚本引擎的独立的Java类装载器，改为在Java应用中通过`GroovyClassLoader`直接加载指令处理DSL的脚本。

我们尝试过的几种方案都有一处共同的缺点，这与Groovy元编程概念的用法有关。本节我们将继续改进，尝试建立更好的Groovy元编程模型来驱动DSL。

5.3.2　控制元编程的作用域

在前面的方案中，我们向已有的Groovy类中注入方法，做法是在相应类的`MetaClass`中增加方法。



Groovy知识点

- **`ExpandoMetaClass`及其元编程原理**。`ExpandoMetaClass`是一种特殊的Groovy元编程结构，允许使用者通过简洁的闭包语法，动态地增加方法、构造器、属性和静态方法。
- **闭包（closure）和委托（delegate）**。Groovy语言的闭包是在一个地方定义，在另一个地方执行的lambda，用法很像Ruby的块（block）。委托对象一般是闭包所从属的对象，但可在运行时更改。
- Groovy语言的**类声明**。类似于Java，但可省去繁琐的类型声明，且Groovy的语法较简洁。
- Groovy语言的**Category特性如何控制元编程的作用域**。Category是Groovy语言除`ExpandoMetaClass`之外的另一种元编程手段。程序中对元对象的改动可以通过Category控制其作用范围。

我们在代码清单3-1里面，用了下面两行代码向`Integer`类注入`shares`和`of`方法：

```
Integer.metaClass.getShares = { -> delegate }
Integer.metaClass.of = { instrument ->  [instrument, delegate] }
```

因为做了这样的铺垫，我们才得以写出下面的DSL脚本（取自代码清单3-2）：

```
newOrder.to.buy(100.shares.of('IBM')) {
  limitPrice   300
  allOrNone    true
  valueAs      {qty, unitPrice -> qty * unitPrice - 500}
}
```

我们进行注入的时候利用了Groovy的`ExpandoMetaClass`，通过它可以在运行时向已有的类中添加方法、属性、构造器以及静态方法。但`ExpandoMetaClass`的问题是，注入到类中的属性和方法是全局有效的，程序会改变JVM内所有线程中的全部类实例的行为。`ExpandoMetaClass`把你对类的改动公开给所有用户。但凡像这样可能波及其他人和其他程序的情况，都应该三思而后行。Ruby的猴子补丁也有同样的全局作用问题，一样会对其他用户产生附带的影响，而且不同用户对于类和方法的定义有着不同的预期，彼此之间可能不相容。

Groovy拥有对元编程作用域进行细粒度控制的能力，这是你在实现Groovy DSL时应该充分利用的一个特点。因为这项特性的重要性，我们会单独用一节的篇幅来讨论Groovy实现。

1. Groovy的元对象协议和Category特性

Groovy的元对象协议（MOP）是另一种注入手段，可以有选择、有节制地对已有类进行注入。用这种方式注入的属性并不会完全暴露给全局，而是将其作用域限制在一定范围的代码块之

内。程序员在一种称为Category的特殊类中定义准备注入的新方法。Category在Groovy DSL的创作中应用得非常普遍。（关于Groovy语言Category特性的详细解释请参阅5.7节文献[2]。）下面我们就要利用Category特性来改造指令处理DSL了。代码清单5-10用Groovy语言对`Order`的基本抽象进行了建模。

代码清单5-10　Groovy语言实现的`Order`类

```
class Order {
  def name
  def quantity
  def allOrNone = false
  def limitPrice
  def valueClosure

  def Order(stockName, qty) {
    name = stockName
    quantity = qty
  }

  def limitPrice(price) {limitPrice = price}

  def allOrNone() {allOrNone = true}

  def valueAs(closure) {
    valueClosure = closure.clone()          ◁ 确保线程安全
    valueClosure.delegate =
      [qty: quantity, unitPrice: limitPrice]          ◁ 绑定自由变量
  }

  String toString() {
    "stock: $name, number of shares: $quantity,
    allOrNone: $allOrNone, limitPrice: $limitPrice,
    valueAs: ${valueClosure()}"
  }
}
```

我们的DSL考虑让用户使用自然的陈述方式来表达他希望买入或卖出的股票数量，例如类似“`200.IBM.shares`”的写法。我们会运用Groovy语言的Category特性来实现这种表达方式，不过首先需要一个辅助类的帮忙。下面定义的`Stock`类是对此表达形式的抽象，指令的其余信息可以放在一个闭包里传递给它。

```
class Stock {
  def order

  Stock(orderObject) {
    order = orderObject
  }

  def shares(closure) {
    closure = closure.clone()          ◁ 确保线程安全
    closure.delegate = order          ◁ 将指令相关信息交给委托对象
    closure()
    order
  }
}
```

5

与其直接说明下一步如何实现，倒不如先让你看看改造完成之后的指令处理DSL是什么样子，到时候你胸有成竹，才更容易理清实现的思路。老鲍将使用这样的脚本来发出他的股票交易指令：

代码清单5-11　一段指令处理DSL的脚本

```
buy 200.GOOG.shares {
    limitPrice 300
    allOrNone()
    valueAs {qty * unitPrice - 500}
}

buy 200.IBM.shares {
    limitPrice 300
    allOrNone()
    valueAs {qty * unitPrice - 500}
}

buy 200.MSOFT.shares {
    limitPrice 300
    allOrNone()
    valueAs {qty * unitPrice - 500}
}
```

从中可以看出我们需要给`Integer`类加上一些方法，这一次我们要通过Category来实现。

2. 基本的DSL

下面是第一个Category的代码，它的作用是帮我们构建不同的`Stock`实例。

代码清单5-12　通过Category在`Integer`上增加方法

```
class StockCategory {
  static Stock getGOOG(Integer self) {
    new Stock(new Order("GOOG", self))
  }

  static Stock getIBM(Integer self) {
    new Stock(new Order("IBM", self))
  }

  static Stock getMSOFT(Integer self) {
    new Stock(new Order("MSOFT", self))
  }
}
```

由这段`StockCategory`的定义可知，调用`200.IBM`将返回一个`Stock`实例。然后我们在`Stock`实例上调用`shares`方法，把别的指令详情放在一个闭包里，作为参数传递给`shares`方法。在`Stock`类的定义中，将`shares`所接收闭包的委托对象设置为一个`order`实例。这样，代码清单5-11中出现的`limitPrice`、`allOrNone`、`valueAs`的上下文就都有了着落。在现实的项目中，代码清单可以根据数据库中的股票列表自动生成。

至此指令处理DSL的基本引擎已经就绪，我们还可以在最后加上一个Category让脚本更有条理，然后装好Java启动入口就算顺利完工。

5.3.3 收尾工作

请你再看一遍代码清单5-11。脚本中每一条指令都以buy开头，也就是说，我们需要向Groovy的Script类注入一个buy方法。我们依旧用Category来实现：

```
class OrderCategory {
  static void buy(Script self, Order o) {
    println "Buy: $o"
  }

  static void sell(Script self, Order o) {
    println "Sell: $o"
  }
}
```

作为演示，我们只是简单地将用户输入的指令打印出来，供检查各属性是否正确设置。在实际的项目中，这个地方应该替换为有意义的相关领域逻辑。

做完这一步，Groovy的DSL实现任务就完成了。现在只需要准备一个负责执行DSL脚本的执行器，供Java应用程序调用。执行DSL的Groovy代码如下，其中导入了之前定义的两个Category：

```
class DslRunner {
  static runDSL(dsl) {                           ❶ 导入相关Category的定义
    use(OrderCategory, StockCategory) {        <-
      new GroovyClassLoader().parseClass(dsl as File).newInstance().run()
    }
  }
}
```

注意代码中的`use {}`块❶，我们通过Category注入到现有类的方法，仅在这个块的作用域内有效。最后我们在Java应用程序内调用`DslRunner`：

```
public class LaunchFromJava {
  public static void main(String[] args) {
    DslRunner.runDSL("newOrder.dsl");
  }
}
```

大功告成！一段Groovy的DSL实现就在你眼前化蛹为蝶。图5-9形象地说明了DSL脚本被翻译为语义模型，然后被送入执行阶段的过程。

Groovy DSL的进化之旅到这里告一段落。下一节，我们将发挥Clojure的特长，实现一种完全不同风格的DSL。

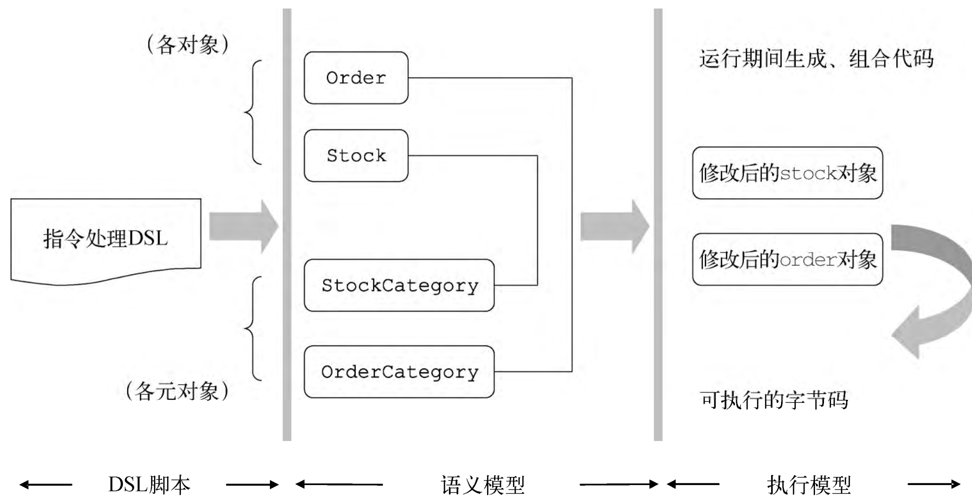


图5-9　从Groovy DSL脚本到语义模型，再到执行模型的转化过程

5.4　思路迥异的 Clojure 实现

本节将为你展示如何在Clojure语言中实现计算交易的现金价值。（假如记不清交易的现金价值是如何定义的，请翻阅4.2.2节的插入栏。）我们照例采用基于DSL的实现方式，自底向上先建立若干小的领域抽象，然后利用Clojure的组合子把它们组合起来。

同样的用例已经在5.2.4节中用Ruby语言实现过一遍，那我们为什么要重复呢？因为Ruby语言的编程范式与Clojure完全不一样。Ruby是一种面向对象语言，运行时元编程是其主要的DSL实现手段。Clojure基本是一种函数式语言，它的宏特性具有很强的编译时元编程能力。显然Clojure的实现思路会与Ruby或Groovy有很大差异。即使用例相同，基于Clojure的DSL和基于Ruby的DSL实现起来完全可能是两回事。所以，我们在此特意选择与Ruby范例相同的用例，向你说明选择不同宿主语言对设计决策的影响。表5-3列举了Clojure相对于Ruby的关键差异点。对Clojure语言本身的详细介绍，请参阅5.7节文献[6]。

表5-3　用Clojure语言实现DSL的时候需要改变思维方向

Ruby语言的DSL实现	Clojure语言的DSL实现
以对象和模块为思考对象，考虑如何通过元编程把运行时的对象和模块组合在一起	以用例中的函数为思考对象，考虑如何通过Clojure序列（sequences）的lambda操作来组织各函数
运用method_missing、const_missing等技巧，以及其他动态元编程特性使DSL简洁有表现力	运用宏将DSL语法转换为一般的Clojure语法成分，转换完全在编译时完成
Ruby或Groovy语言实现的DSL，其语法风格不一定接近于宿主语言	Clojure语言实现的DSL，因为仍然以“S表达式”为结构基础，其语法风格接近于Clojure

我强烈建议你先把前面的Ruby实现温习一遍，再跟着我一起学习下面的Clojure实现，这样你会对两者的差异有更深的体会。

5.4.1 建立领域对象

构建DSL先要有一些基本抽象作为核心的领域模型。那么，我们第一步先设计好“交易”抽象，并且给它定义一个工厂方法（见代码清单5-13），然后根据传入的信息生成交易对象。

定义 工厂方法是一种设计模式，它为一组相似对象的实例创建操作提供单一的交互入口。



Clojure知识点

- **基本的函数定义和语法**。Clojure的语法接近于Lisp，陌生的前缀表示法可能让你措手不及。如果不习惯Clojure的语法，请细读一下5.7节文献[4]的基础知识。
- **定义Map数据结构**。Clojure编程中常用Map数据结构来仿造“类”这种OO编程结构。

属性可以来自Web请求、文本文件、数据库，系统连接到的任何数据源都可以作为交易对象的信息来源。工厂方法从请求中提取信息，并建立一个map来表示一笔交易的全部属性。

代码清单5-13 生成交易的Clojure代码

```
(defn trade
  "根据请求产生一笔交易"
  [request]
  {:ref-no (:ref-no request)                                          ❶ 从请求构建交易
   :account (:account request)
   :instrument (:instrument request)
   :principal (* (:unit-price request) (:quantity request))           基本价值 = 单价 * 数量
   :tax-fees {}})                                                     ❷ 税费的具体内容稍后再填入

(def request
  {:ref-no "trd-123"                                                  ❸ 请求样例
   :account "nomura-123"
   :instrument "IBM"
   :unit-price 120
   :quantity 300})
```

Clojure在对象系统之上向用户呈现了一个函数式的编程模型。例中我们将抽象实现为一系列键-值对，也就是Clojure `Map`的形式。其中`trade`是一个函数，负责根据从`request`输入的相关信息建立必要的抽象。作为输入的`request`❸也是一个`Map`，被实现为各键的函数。当我们从`Map`中提取信息的时候，所用语法与一般的函数调用相同❶。例如字面量语句`(:account request)`意为从`Map`中取出`account`键的值。

`trade`方法清晰地表达了领域意图和领域语义。Clojure的map字面量语法起到了命名参数的效果，领域概念可以直接被映射为编程元素，从而提高了代码的表现力。`tax-fees`这个map目

前只是一个占位符❷，下一节对生成的交易进行后续充实操作时再填入具体的内容。

5.4.2　通过装饰器充实领域对象

下一步要对基本抽象进行补充，使之适用于交易周期中的具体用例。新的特性以装饰器的方式附加到基本抽象之上，我们在5.2.4节就通过同样的模式在Ruby实现中加入税费组件。不过这一次的实现手段是Clojure语言的编译时元编程和宏。



DSL的设计工作要将目标语法映射到语言背后的语义。那么当实现语言变了，思维方式也应该随之改变。



Clojure知识点

- **高阶函数**。Clojure支持高阶函数特性，函数完全可以像值一样使用。函数可以作为参数来传递，也可以充当返回值，诸如此类。
- **宏**是用Clojure语言开发DSL的根本秘诀。宏是编译时元编程的基本组织元素。
- **“Let绑定”和词法作用域**。不管作用域有多小，你都可以根据需要精确指定绑定的作用域。
- 掌握**Clojure标准库函数**。Clojure网站（http://clojure.org）上有丰富的相关资源。
- **不可变数据结构**。Clojure提供不可变、持久化（persistent）的数据结构。这里的“持久化”指即使改动了数据结构之后，用户仍可以访问所有旧版本的数据。详见5.7节文献[4]。
- **若干基本的组合子**，如`reduce`和`->`。组合子是以其他函数作为参数的函数，能帮你写出简洁而有表现力的代码。

怎么样才能动态地给抽象增加新行为，却不增加运行时的性能负担呢？Clojure给出的答案是编译时mixin。我们来看看具体的做法。

1. 使用Clojure组合子

假设我们有一个`with-tax-fee`结构，它的作用是在现有的Clojure函数上引入新的行为，从而给我们的交易加上税费。在下面的代码片段中，当`with-tax-fee`作用于`trade`函数时，我们会得到一个新的`trade`函数，新函数在原有属性集合上增加了`:tax`和`:commission`两则映射。

```
(with-tax-fee trade
 (with-values :tax 12)
 (with-values :commission 23))
```

在这里，`with-tax-fee`充当了`trade`函数的装饰器。现在当你根据`request`执行`trade`函数时,其中的税金和佣金项目将分别被设置为基本价值的12%和23%。（税金和佣金一般以交易基本价值的百分比来计算。）

除了DSL的实现者，别的人一般不需要关心`with-tax-fee`、`with-values`等语言构造的具体实现，把它们当做一般的组合子用于交易DSL的抽象设计即可。不过本节既然讨论DSL的实现，

自然应该探讨，什么样的函数才能充当另一个函数的装饰器，为其补充新的行为。下面是with-values的实现。

代码清单5-14 给trade抽象包裹一层新行为

```
(defn with-values [trade tax-fee value]        ❶ 高阶函数
  (fn [request]                                 ❷ 返回一个函数
    (let [trdval (trade request)                ❸ 获取交易价值
          principal (:principal trdval)]
      (assoc-in trdval [:tax-fees tax-fee]      ❹ 以基本价值的百分比的形式存入:tax-fees Map
               (* principal (/ value 100))))))
```

with-values组合子对trade函数的输出做了不少补充工作。虽说本书的主题不是介绍Clojure语言，但对于这段代码，有必要深入剖析Clojure语言特性在其中所起的作用。这样有助于你理解Clojure语言如何以其抽象能力化繁为简，向用户呈现简洁的API，如表5-4所示。

表5-4 解剖Clojure API

Clojure语言特性	在DSL中的运用
高阶函数是实现DSL的基本要素之一	with-values的第一个参数是个函数❶；with-values函数返回另一个函数❷；这些都是Clojure语言把函数视同于一般的值的具体表现。Clojure支持高阶函数，所以你可以把函数当成参数来传递，当成返回值来获取，就像语言中的其他数据类型一样
	fn表示一个匿名函数❷。with-values所返回的这个匿名函数，对with-values的输入函数trade进行了增补，增加填充:tax-fees Map的行为
求值时指定词法上下文以控制其作用域	我们用新函数的参数来调用trade❸，然后把tax-fee的值补充到结果的Map里
	将let后面的多个绑定项依先后次序进行绑定；所以后一项绑定principal可以引用前一项绑定trdval
不可变性、实现持久化数据结构的能力	这段代码的最后一步，是把tax-fee和value参数凑成一个键–值对❹，放入trade函数在❸处返回的Map
	在这个过程中，原来的Map保持不变。Clojure实现了不可变和持久化的数据结构。因此，assoc-in每次调用都会返回一个新的Map，比原来的Map多了参数里指定的一对键值
函数可以自然地组合在一起	with-values的返回结果是一个函数,这有利于实现串联调用。我们可以把代码写成这个样子： (with-tax-fee trade (with-values :tax 12) (with-values :commission 23)) 我们把两次with-values调用跟原来的trade函数串联在一起。方法的串联调用体现了语言的组合性，我们说过这是一种优秀的DSL特质。Clojure等语言将函数视同一般的值，造就了很好的组合能力

那么，with-tax-fee怎样与with-values合力构成新的trade函数呢？这是我们下面要讨论的内容。

2. 高阶函数实现的装饰器

我们还欠缺一个知识点才能讲解清楚with-tax-fee的原理，这个知识点虽然不怎么起眼，却是装饰器的实现基础。对比总是围绕着对象来展开的Ruby实现，我们越来越认识到，Clojure

把函数放在了更重要的位置上，而且它为此准备了很多巧妙的手段。作为一种基本思路，Clojure DSL就应该从Clojure里发掘最自然、最符合语言习惯的手法来实现。下面的代码片段演示了Clojure语言的函数“串接”（threading）手法。

```
(def trade
    (-> trade
        (with-values :tax 20)
        (with-values :commission 30)))
```

->宏将它的第一个参数传给参数序列中的下一个form，结果再传给再下一个form，如此依次传递下去，如同把这些form串成了一串。Clojure的函数串接让实现装饰器变得轻而易举，因为只要用->把一个函数串接到它的装饰器，就等于完成了对函数的重定义。代码清单5-15所示的装饰器实现代码就运用了这种技巧。这段代码来自Clojure语言Web开发框架Compojure（详见http://github.com/weavejester/compojure）。如果不熟悉Clojure语言，你可能要费一些功夫才能理解这里提到的编程概念。但一旦领会了Clojure用小的函数式抽象组合成大的函数式抽象的设计思路，你就能欣赏到上面短短四行代码的强烈美感，并且感激它们对DSL实现所做的贡献。

3. 画龙点睛的Clojure宏

与其让用户自己串连装饰器，不如用一个宏把函数串接操作包装起来，除了简明易懂外，还不产生任何运行时的额外负担。于是就有了代码清单5-15。

代码清单5-15　Clojure装饰器

```
(defmacro redef
  "重定义一个现有值，保持元数据不变。"
  [name value]
  `(let [m# (meta #'~name)
         v# (def ~name ~value)]
     (alter-meta! v# merge m#)
     v#))

(defmacro with-tax-fee                                  ❶ Clojure宏
  "将函数包入一个或多个装饰器。"
  [func & decorators]
  `(redef ~func (-> ~func ~@decorators)))
```

寥寥几行代码就勾勒出with-tax-fee的全貌——一个Clojure语言写就的，以非侵入方式向现有抽象增补新行为的编译时装饰器。with-tax-fee被实现为一个宏❶，因此它所封装的代码将在编译过程的宏展开阶段被释放出来。

在装饰过程中，输入函数的原值，即根绑定（root binding）会被我们重新定义，同时保持其元数据不变。这就是redef宏的工作。这个装饰过程不像Ruby元编程那样在执行阶段才实际发生；Clojure在运行时不存在任何元对象，所有的定义在宏展开阶段就已经确定了。

现在我们的DSL可以在交易抽象上补充税费计算逻辑了，这着实费了不少功夫。有了新的带装饰的trade函数，计算交易的现金价值的API也很容易能定义出来。我们之所以能实现有意义

的领域抽象，讨论至今的Clojure特性功不可没。下面的API明白无误地说明了自己是如何计算交易净值的。任何熟悉金融交易和领域语言的人都能理解该函数的意图。

```
(defn net-value [trade]
 (let [principal (:principal trade)
       tax-fees (vals (trade :tax-fees))]
   (reduce + (conj tax-fees principal))))
```

❶ 组合子提高了抽象程度

这几行语句是对Clojure语言简洁性的最好证明。Clojure是一种紧凑的语言，适合在比较高的抽象层次上进行编程。上面最后一行代码用非常简练的语言描述了复杂的操作。reduce组合子递归地作用于后面的序列，依次施加指定的函数（+）❶。

4. 我们的成果

我们的DSL只剩下最后的执行步骤，成功在望，反而不必急于一时。我们不妨耐心对照表5-5，总结一下到目前为止我们为实现计算交易现金价值的DSL做了哪些事情。

表5-5　DSL的演变过程

DSL的演变步骤	实现细节
(1) 设计交易的基本抽象	通过工厂方法trade完成以下工作：
	(1) 从外部数据源接收数据
	(2) 生成交易对象，对象表示成Clojure Map的形式
(2) 向领域对象注入新行为	新的trade函数为了计算现金价值而被注入了税费方面的行为
采用以下技巧：	**如何将税费数据填充到交易中：**
■ 装饰器模式	(1) 定义with-values函数，对trade函数的输出进行增补，加入新行为
■ Clojure宏	(2) 通过装饰器模式将税费数据填入trade函数的输出
	(3) 定义with-tax-fee宏，该宏可将with-values多次应用于一个现有
	注意：with-tax-fee使用了编译时元编程技术，没有额外的运行负担函数
	net-value函数除了接收第2步修改后的trade函数，还将完成以下工作：
(3) 定义net-valu函数，计算交易的现金价值	(1) 从交易信息中取得基本价值
	(2) 从交易信息中取得税费数据
	(3) 按照具体的领域逻辑计算现金净值

Clojure语言的设计理念不同于Ruby、Groovy、Java等语言。尽管扎根在Java的对象系统之上，它的语言习惯却是函数式的。Clojure编程不能沿用面向对象的思路。纵观本节的实现，我们其实并没有运用什么特殊的技法来设计DSL，你看到的全都是自然的Clojure编程风格。

图5-10描绘了Clojure DSL脚本的生命周期，请你多看两眼，加深理解。

感觉疲倦了吗？其实，我们还有一件关于Clojure DSL的事情留着没说，那就是Clojure REPL（read-eval-print-loop）这个让你能够即时观察、调整DSL的互动窗口。小憩一下，补充点能量，接下来是互动环节，少了活力可不行。我们会让你直接与Clojure解释器面对面地交流，实地运行本节刚刚设计好的DSL。

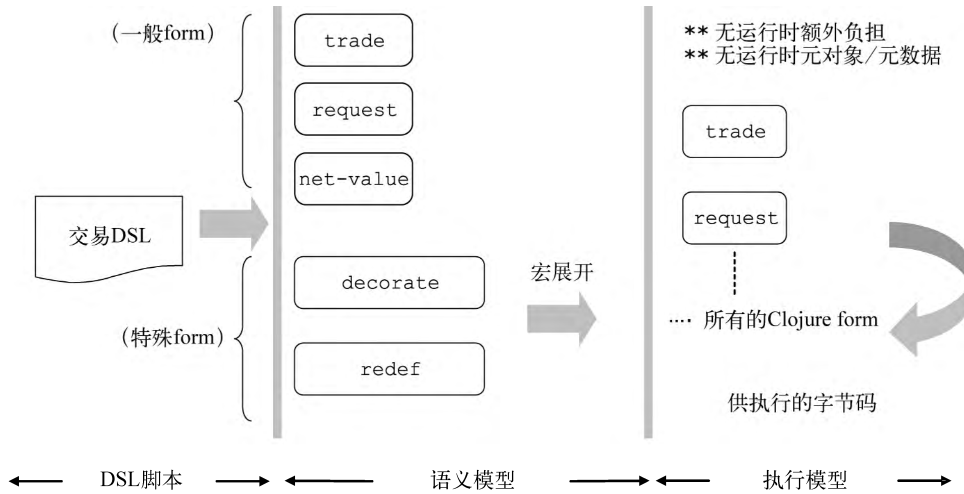


图5-10　从Clojure DSL脚本到Clojure执行模型。留心观察DSL脚本被执行之前经历的一系列准备。我们在第1章就提过，语义模型是DSL脚本与执行模型之间的桥梁

5.4.3　通过REPL进行的DSL会话

Clojure等动态语言允许你直接通过一个REPL界面与语言运行时交互。（关于REPL的介绍请阅读http://en.wikipedia.org/wiki/Read-eval-print_loop）。REPL让你实时观察DSL，即时修改其行为并看到修改的效果。这项特性绝对是实现DSL无缝改进的有力帮手。

我们的DSL的简洁度和表现力都达到了相当高的水平，举例来说，用我们的DSL表达现金价值的计算逻辑，只需要写`(net-value (trade request))`这么简单的一句话。你可以即时创建一笔交易，放在REPL里运行，然后修改`trade`函数，以装饰器的形式增加业务规则。下面是用我们的DSL在Clojure REPL中进行的一段会话：

```
user> (def request {:ref-no "r-123", :account "a-123",
                    :instrument "i-123", :unit-price 20,
                    :quantity 100})
#'user/request

user> (trade request)
{:ref-no "r-123", :account "a-123", :instrument "i-123",
  :principal 2000, :tax-fees {}}

user> (with-tax-fee trade
        (with-values :tax 12)
        (with-values :commission 23))
#'user/trade

user> (trade request)
{:ref-no "r-123", :account "a-123", :instrument "i-123",
  :principal 2000, :tax-fees {:commission 460, :tax 240}}

user> (with-tax-fee trade
```

```
        (with-values :vat 12))
#'user/trade

user> (trade request)
{:ref-no "r-123", :account "a-123", :instrument "i-123",
  :principal 2000, :tax-fees {:vat 240, :commission 460, :tax 240}}

user> (net-value (trade request))
2940
```

好的DSL应该对外呈现易于使用、充分体现领域语汇精髓的API界面，同时将其复杂的实现隐藏在API之后。这是优秀DSL的决定性品质之一。本节的Clojure REPL交互真实地展现了DSL的简洁度。我们的DSL始终让你觉得它处处呼应了交易员的实际工作用语，虽然是领域语言，却能刚好用Clojure语言来实现。

一名合格的设计者在学习任何新范式的时候，都不要忘记掌握它的缺陷。我们已经讲解了不少正面的模式、惯用法和最佳实践，为你指明DSL的实现道路。下一节要说说反面的缺陷，提醒你避开途中的险阻。

5.5 告诫

一直以来，本章向你展示的都是正面的例子。我们用了三种最流行的动态JVM语言来讨论DSL的实现，不但让你见识了不同语言的各种惯用法和实现技巧，还让你亲手实现了若干证券交易应用领域的实用DSL脚本。然而，不管目前进展得多么顺利，你终究会遇到一些陷阱，而有些潜在的危险我必须给你指出来。

我特意为本章分属三种语言的例子选取了关系密切的三个用例。这么做的目的，是强调即使问题相同，你也应该根据不同语言的特点，选择不同的解决手段。在Ruby实现中适合用动态元编程来解决的问题，使用Clojure时，就不一定能照搬Ruby的套路。你必须学会选择正确的工具去做正确的事。而恰恰在你选择的过程当中，很容易冷不防被常见的陷阱绊倒。我们打算从DSL开发的角度讨论其中的一些陷阱。

5.5.1 遵从最低复杂度原则

实现内部DSL的时候，应该选择宿主语言中最简单、同时最适用于解答域模型的惯用法。我们经常可以看到开发者在不必要的情况下选用元编程手段,例如Ruby的猴子补丁就是一个被滥用的典型。（还记得猴子补丁吗？猴子补丁可以打开一个类，修改其中的方法和属性。由于修改结果会作用于全局，Ruby的猴子补丁特别危险。）很多时候Ruby模块足以代替猴子补丁，与其打开一个类并插入新方法，不如把方法放入一个`Module`中，然后有针对性地将`Module`包含进有需要的类中。

5.5.2 追求适度的表现力

过度追求DSL的表现力，有可能给实现带来无谓的复杂性。语言的表现力能满足用户要求就

够了。下面的Ruby DSL片段来自5.2.2节，对于程序员来说，这样的语言已经足够让人理解其领域语义。

```
TradeDSL.new.new_trade 'T-12435',
  'acc-123', :buy, 100.shares.of('IBM'),
  'unitprice' => 200, 'principal' => 120000, 'tax' => 5000
```

表现力已经够充分了！那么我们为什么要继续开发解释器版DSL呢？有两个理由。首先，我希望表现力提高之后，DSL能得到团队中领域专家的认可。领域专家老鲍是第一位抱怨原版DSL所含非本质复杂性的用户，解释器版更符合他平常在交易台前的用语。其次，我希望充分展示Ruby的动态性的潜力。当你在现实中真正设计DSL时，一定要记住表现力水平应该配合用户的身份。

5.5.3　坚持优秀抽象设计的各项原则

经常会遇到一些现实情况，让你忍不住想要增加DSL的枝节去提高用户的认同感，结果往往就违反了第1章讨论过的优秀抽象的设计原则。语言里的赘词和虚饰多了，封装就容易被破坏，实现的内部细节也更容易暴露。为提高DSL的表现力而削弱抽象的不可变性，并不一定是划算的。这方面的取舍可以看代码清单5-5的例子。我们把`Instrument`抽象做成可变的，得以将票据创建逻辑表达为流畅的DSL语句。然后在代码清单5-7中，我们进一步利用抽象的可变性提高DSL的表现力。

这里的告诫并不是让你放弃对表现力的追求。请你记住，语言设计是一种充满了取舍和妥协的工作。任何决策、对抽象设计原则的任何让步，都应该经过审慎评估。对设计原则的调整，也应该以DSL目标用户的身份背景为标杆。

5.5.4　避免语言间的摩擦

按照一般的认识，不同的DSL不能组合使用。一种DSL总是针对一个专门的领域。你在设计交易系统的DSL时，总是以建模领域为参照去调整其表达方式。至于如何与帐务明细DSL、投资组合管理DSL之类的第三方DSL集成，你根本想不了那么多。

虽然你无法预料一切情况，但设计中还是应该尽量提高抽象的组合能力。函数天生比对象容易组合。如果你的实现语言支持Ruby、Groovy、Clojure中的高阶函数，那么应该把设计重点放在组合子的串联上，通过组合子之间的联系，自然凝聚为语言的脉络。可组合的抽象具有诸多优点，有利于并发，具体的讨论请查阅附录A。

如果你的抽象不能组合，DSL就成了一盘散沙。语言成分一个个孤立着零落不成句，领域用户又怎么可能用得自然。

以上就是DSL设计中最容易踩中的几个陷阱，务必加以注意。从语言中挑选哪一部分子集用于DSL实现，是极端重要的设计决策。你要时时记住DSL的集成需求，始终尊重优秀抽象的设计原则。

5.6　小结

祝贺你！用动态类型语言实现内部DSL的长篇讨论就要结束了。Ruby、Groovy和Clojure语言作为JVM平台语言多样性的代表，被我选为讲解用的实现语言。

JRuby是Ruby语言的Java实现，充当了Ruby语言与Java对象模型互操作的桥梁。它既有Ruby的强大元编程能力，又得益于Java的互操作性。Groovy语言本身就被当做一种Java DSL，与Java共用相同的对象模型。Clojure语言虽然也建立在Java的对象模型之上，却提供了Lisp那种强烈的函数式编程范式。

本章引导你使用以上三种语言实现了若干典型、现实的交易系统用例。Ruby的元编程能力强，可以使DSL在运行时保持动态，所以你可以组织建造高阶抽象。Groovy的运行时能力与Ruby相似，但它与Java的互操作更严丝合缝，毕竟两者共享同一个对象模型。

我们从第2章开始设计的指令处理DSL迎来了用Groovy语言实现的最终版本。通过这个例子，你应该了解了一般DSL的增量式演进的迭代过程。Clojure是运行在JVM上的Lisp语言，拥有出类拔萃的编译时元编程能力，也就是所谓的宏。你学习了如何运用宏来提高DSL的表现力和简洁度，并且知道它不会像其他语言的元对象协议那样增加运行时的负担。

最后，只要你能记住每个设计决策意味着哪些妥协和代价，肯定能做好DSL的设计。说到底，语言设计工作就是要检验你能不能在表现力和实现代价之间找好平衡。对于DSL来说，它充当着开发者和领域专家之间沟通渠道的角色，因此能充分传达代码的意图才是最根本的追求。

要点与最佳实践

- 设计内部DSL时应该掌握**所有的Ruby元编程手段**。不要忘记元编程有代码复杂性和性能两方面的代价。
- **优先选用**Groovy Category代替`ExpandoMetaClass`来控制元编程的作用域。
- Ruby的**猴子补丁**很吸引人，但它作用于全局命名空间。在DSL实现中使用猴子补丁时要三思而后行。
- Clojure**虽然在Java平台上实现，却是一种函数式语言**。Clojure DSL的设计应该围绕领域函数进行。充分发挥函数式编程的长处，运用高阶函数和闭包来设计DSL的语义模型。

与三种最流行、最动态的JVM语言结伴同行的DSL设计之旅到了终点。经过这番历练，想必你已经对实现DSL的各种惯用手法有了基本的概念。能否选择符合语言习惯的正确实现方案，对开发工作有着决定性的影响，你选择的方案决定了DSL API的表现力水平。学习完本章，你对DSL开发的掌握程度又上了一个台阶，具备了深入探索Ruby、Groovy和Clojure语言各自实现技巧的基础。

下一章我们将跨过隔开类型系统两大阵营的那道篱笆，从另一端着眼，观察静态类型会塑造出什么样的DSL实现。等着你的还有充满趣味的练习——用Scala语言开发内部DSL。

5.7 参考文献

[1] Thomas, Dave, Chad Fowler, and Andy Hunt. 2009. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*, Third Edition. The Pragmatic Bookshelf.

[2] Subramaniam, Venkat. 2008. *Programming Groovy: Dynamic Productivity for the Java Developer*. The Pragmatic Bookshelf.

[3] Perrotta, Paolo. 2010. *Metaprogramming Ruby: Program Like the Ruby Pros*. The Pragmatic Bookshelf.

[4] Halloway, Stuart. 2009. *Programming Clojure*. The Pragmatic Bookshelf.

[5] Abelson, Harold, Gerald Jay Sussman and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs*, Second Edition. The MIT Press.

第 6 章

Scala语言中的内部DSL设计

6

本章内容

- ❑ 对Scala语言本身的介绍
- ❑ 用Scala语言开发内部DSL
- ❑ 组合多个DSL
- ❑ 运用Monad化结构

以DSL驱动的开发方式有其擅长和不擅长的方面，前面几章说了不少。现在你肯定已经认识到，对于应用中反映业务规则的部分，DSL能非常有效地疏通开发团队与领域专家团队的沟通渠道。上一章讨论了使用几种动态JVM语言担当内部DSL宿主的能力。这一章要介绍的是一种非常具有这方面潜力的静态JVM语言——Scala。

本章继续重点讨论具体的实现，直接与编程相关的实践性内容将占据较大篇幅。我们首先论证Scala语言是一种称职的内部DSL宿主语言，然后在真实的DSL设计中检验这个结论。图6-1是本章讨论进程的路线图。

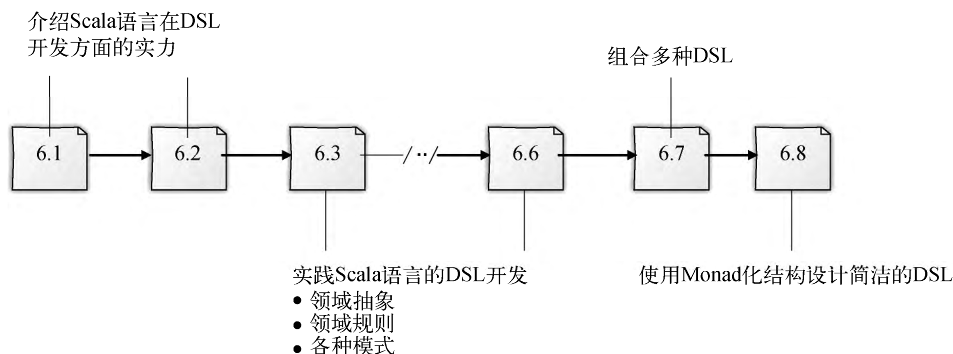


图6-1　本章路线图

6.1节和6.2节将确立Scala作为DSL宿主的资格。在此之后，我们就证券交易后台系统中的真

实用例展开详细讨论。6.3节和6.6节会实际演练许多惯用法、最佳实践和模式。6.7节讨论如何将多种DSL组合成更大规模的语言。本章最后介绍Monad在DSL设计中的作用，它能塑造出简洁、带有函数式风格、富有表现力的DSL。

学完本章，你将全面掌握如何在Scala这种宿主语言中设计DSL。你会学到如何建模领域组件，然后围绕它们建立简单易用、语义丰满的语言抽象，熟悉这方面的惯用法和最佳实践。怎么样，想学吗？我们开始吧。



本章代码片段都以Scala 2.8版本为准。不熟悉Scala语法的读者可以参阅附录D的Scala语法速查表。

6.1 为何选择 Scala

Scala同面向对象语言一样，拥有丰富全面的抽象机制，塑造了DSL的简洁度和表现力。DSL不能脱离其实现模型而独立存在，我们说过，DSL是罩在实现模型外面的一层门面。本章将分析Scala发展为宿主语言的历程，围绕设计底层模型和外层DSL两方面展开。表6-1列举了Scala常用于DSL设计的一些代表性语言特性。

表6-1 用于DSL设计的代表性Scala特性

语言特性	Scala的具体做法
灵活的语法	Scala的表层语法简练，有许多手段可使DSL更贴近真实的领域用语 **例如：** ■ 可省略方法调用的点符号 ■ 分号推断 ■ 中缀运算符 ■ 可省略方法调用的圆括号
可扩展的对象系统	Scala是面向对象的。它与Java使用相同的对象模型，然后利用自身先进的类型系统，在很多方面进行了扩展 **Scala的对象语义：** ■ trait的用途是基于mixin的实现继承（见6.10节文献[12]） ■ 抽象类型成员和泛型类型参数这两项语言特性都可以使类具有正交扩展能力（见6.10节文献[13]） ■ 通过自类型标注（self-type annotation）对抽象的正交扩展进行约束（见6.10节文献[14]） ■ Case类的用途是实现值对象[1]
函数式编程能力	Scala是一种多范式的编程语言，结合了面向对象和函数式编程的语言特性 **为何采取面向对象与函数式相结合的方式？** ■ Scala中的函数是第一类值；从类型系统层面开始，就支持高阶函数。用户可将自定义DSL控制结构写成闭包，然后当做一般的数据类型传递 ■ 在纯粹的面向对象语言中，一切事物都要套入类的设计形式，不管它本来的领域含义应该是名词还是动词。Scala混合了面向对象与函数特性，更有利于模型贴近问题域的语义

（续）

语言特性	Scala的具体做法
静态检查的鸭子类型	Scala通过结构化类型定义（structural types，见6.10节文献[2]）支持鸭子类型 **与Ruby鸭子类型的差别：** Scala的鸭子类型是**静态检查**的
限定了词法作用域的开放类	Scala通过它的implicit语言结构取得开放类的效果，3.2节的补充内容“Scala implicits”中有相关介绍 **与Ruby猴子补丁的差别：** Scala的implicits结构有词法作用域的限制；通过隐式转换方式添加的行为，需要明确导入具体的词法作用域才生效（详见6.10节文献[2]）
隐含参数	API调用中可以省略部分参数，让编译器去推断。这样做可以精简语法，提高DSL脚本的可读性
模块化的对象组合方式	有独特的对象概念，可用来定义具体的DSL模块。你可以将一些DSL结构定义为抽象成员，推迟进行具体实现

[1] 普通类和case类的主要区别在于，case类的构造函数调用更加简单，可以使用默认的相等性语义，以及支持模式匹配。

这些特性汇集在一起，使Scala成为一种非常适用于内部DSL设计的JVM语言。不过它毕竟是一种新语言，难免令人犹豫。在团队里引入一种新语言，从技术和文化方面来说，都是很大的挑战。公司可能已经在JVM平台上投入巨大资源，相当数量的客户应用也运行于Java平台，程序员群体也花费了无数精力去熟悉各式各样的Java框架。你能为了迎接一种新的编程语言，而放弃多年的积累吗？幸好，Scala允许你循序渐进地完成转换。请看下一节。

6.2 迈向 Scala DSL 的第一步

Scala是一种很好的内部DSL宿主语言，但单凭这一点不足以说服经理在开发环境中引入Scala这种新技术。任何新技术都要控制引入的节奏，以免增加混乱的风险。一般可以先在不太关键的业务上采用，再慢慢普及。

置身于JVM之上，拥有与Java互操作的能力，这是Scala的重要优势。企业可以一边保留Java方面的投入，一边向Scala过渡。Scala DSL的第一步可以有很多走法，在保持基本Java抽象不变的前提下，已经有不少发挥空间。图6-2给出了一些策略，可以供你的开发团队参考。

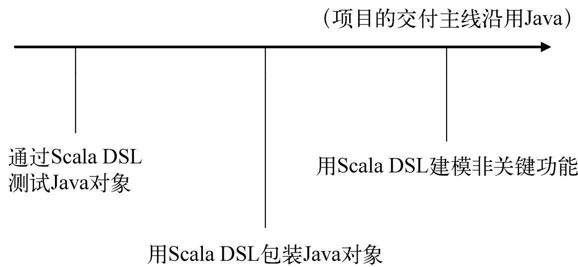


图6-2 Scala不需要一开始就应用到生产代码之中。这里列举了一些学步的方案，选择何种顺序你说了算

从图6-2中可以看出，项目的交付主线可以继续沿用Java，团队中的部分成员在辅助性工作中接触Scala。下面几节将详细说明图中的几种起步方式。

6.2.1　通过Scala DSL测试Java对象

测试是开发过程中的核心任务之一，同时它的技术和框架都有很大的选择空间。测试套件的重要性不亚于生产代码库。业内人士正竭尽全力，希望将测试套件表达得更到位、更详尽。

DSL已经成为测试框架必不可少的一部分。你只要选择一种基于Scala DSL的测试框架，就能立即开始学习用Scala语言设计DSL。例如ScalaTest（见6.10节文献[8]）就是这样一种DSL框架，你可以用它编写DSL，对Java和Scala类进行测试。一开始并不要求你掌握Scala类的写法，只要在这类测试框架内重用原有的Java类即可，目的是熟悉一下基于DSL的开发方式和环境。

6.2.2　用Scala DSL作为对Java对象的包装

本书一再提及Scala与Java的完美契合，我们可以利用Scala的这个特点，为Java对象加上一层包装，获得更灵巧、丰满的表达。3.2.2节可作为这种手法的实证，我们借助Scala的力量，把Java对象打扮成一副精明能干的样子，是相当有说服力的。通过对这种手法的实践，你将掌握如何运用Scala语言在Java对象模型上创作DSL，这是一条简单高效的学习途径。

6.2.3　将非关键功能建模为Scala DSL

大型应用常包含一些不太关键的部分。你可以跟客户协商，把其中一些次要部分作为Scala DSL设计的试验田。如果不愿意放弃主要的Java编译模式，那么可以把Scala DSL做成脚本，然后通过Java 6提供的`ScriptEngine`来运行。

下面详述Scala各项特性的具体用法，深入探讨怎样用它们建立领域模型，然后编排成流畅的DSL。

代码提示　*后面的段落含有大量代码片段，我会插入对一些预备知识的说明，解释必要的语言特性，以便读者理解实现中的细微之处。如有需要，还可以查阅本书附录中相应语言的速查表。*

讨论中将继续沿用金融中介领域的例子，逐步应用不同的特性去修补和改良DSL的设计，最后形成一种可以交付使用的完善DSL。这段旅程将充满乐趣。

6.3　正式启程

本章所需的背景知识你已经了解得差不多了，我们回到正题。本章将研究证券交易领域的各种现实用例，观察在Scala实现语言的作用下，如何将用例转化为生动的DSL。

我选择的用例跟之前讨论Ruby、Groovy实现时的用例差不多。这样你把前后的例子互相对照，就不难看出其中思路的变化。即使问题域相同，静态类型语言和动态语言的DSL设计思路也截然不同。首先我们了解一下Scala有哪些特性可以提升DSL语法的表现力。

> **Scala知识点**
>
> - **Scala语言的面向对象特性**。Scala的类和继承结构有多种设计方式。
> - Scala语言拥有**类型推断**特性；它的运算符等同于方法；语法灵活，可省略圆括号和分号。
> - **不可变变量**（immutable variable）有利于设计函数式的抽象。
> - Scala语言的 **case类和case对象**，其特点适用于设计不可变的值对象。
> - Scala的**trait特性**，可用于设计mixin和多继承。

6.3.1 语法层面的表现力

一门语言的语法是富于表现力还是繁冗拖沓，只有一线之隔。非程序员用户觉得表现充分的语法，程序员可能就觉得烦琐到了极点。5.2.3节为交易DSL设计Ruby解释器时就说过这个问题。还记得2.1.2节老鲍的抱怨吗？Java给指令处理DSL强加了一些不必要的语法复杂性，老鲍很有意见。Scala虽然也和Java语言一样是静态类型的，但它对外呈现了较为精简的表面语法，可以减少对DSL的干扰。代码清单6-1展示了一段常见的Scala代码，它的功能是将`ClientAccount`对象放入已存在的一个账户列表。

代码清单6-1 Scala的语法兼具表现力和简洁性

```
val a1 = ClientAccount(no = "acc-123", name = "John J.")
val a2 = ClientAccount(no = "acc-234", name = "Paul M.")

val accounts = List(a1, a2)

val newAccounts =
  ClientAccount(no = "acc-345", name = "Hugh P.") :: accounts

newAccounts drop 1
```

❶ 命名参数和默认参数
❷ 类型推断
❸ 运算符是一个方法
❹ 可省略的圆括号

即使你不熟悉Scala语言，也能毫无困难地看懂这段代码。表6-2列出的几项特性正是代码简明的关键因素。

表6-2 使Scala语法简洁的各项特性，以代码清单6-1为参照

Scala的简洁性来源	对DSL设计的影响
自动推断句末分号	Scala与Java不同，不要求在语句间加上分号作为分隔符，直接降低了对DSL语法的干扰
命名参数和默认参数	`ClientAccount`类实例化时❶用了命名参数。该类被声明为case类（见代码清单6-3），所以其对象的构造语法较为简便。还有部分参数因为已经在类定义中设置了默认值，所以实例化时可以不用写出来。命名参数和默认参数对于提高DSL脚本的可读性有很大帮助
类型推断	用多个账户对象组成列表时，不需要指定结果列表的类型❷，编译器会帮你推断

（续）

Scala的简洁性来源	对DSL设计的影响
运算符等同于方法	我们通过::运算符向accounts列表增加一个ClientAccount对象❸。实际上等同于在List实例上调用方法，即accounts.::(ClientAccount(no = "acc-345", name = "Hugh P."))。写成运算符的形式，同时省略点号（.），这两项措施大大提升了代码片段的可读性
可省略圆括号	我们调用List类的drop方法从列表中删去第一个账户❹。此处代码省略了圆括号，更贴近一般的阅读习惯

本节讨论的只是一些纯语法层面的表面因素。还有其他特性同样对Scala语法的可读性起了正面的作用，包括它强大的集合字面量语法、允许用闭包作为控制抽象等特性，也包括隐含参数等高级特性。本章将逐一介绍这些特性，讨论它们各自在内部DSL设计中的作用。

为了给后面的DSL打好基础，现在我们需要着手准备一些基本的领域抽象。我们的抽象仍然出自前面几章打过不少交道的证券交易领域。一方面不至于浪费前面学到的领域概念，另一方面也便于在对比中学习各种语言的不同实现惯例。

6.3.2　建立领域抽象

设计Scala DSL时，通常需要有一个对象模型作为基本抽象层。然后运用子类型化手段，实现各种模型组件的特化，再将它们与解答域中符合条件的mixin组合起来，构成更大型的抽象。模型中的操作可通过函数抽象来表示，然后用组合子来组织它们。图6-3说明了Scala抽象实现扩展性的方式，它利用了Scala兼具面向对象和函数式功能双重优势的特点。

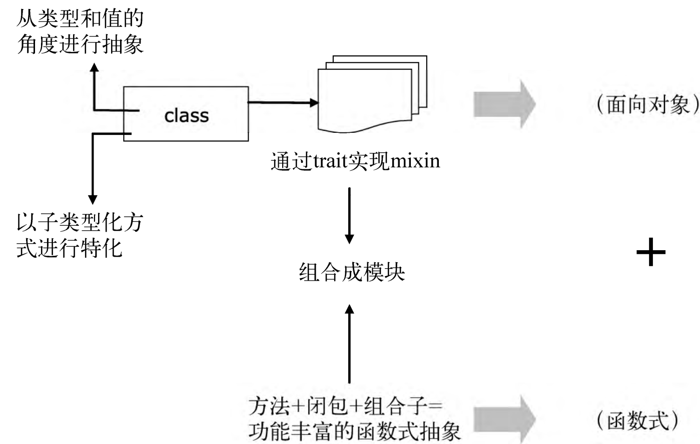


图6-3　Scala兼具面向对象编程和函数式编程功能，两者都可用于建设领域模型。运用Scala的面向对象特性，可以从类型和值的角度进行抽象，以子类型化的方式特化一个组件，然后通过mixin进行组合。而在函数式特性这边，Scala给你准备了高阶函数、闭包、组合子等工具。最后，可以用模块将两方面的成果合并起来，得到最终的抽象

要构建交易DSL的实现，首先从建立问题域的基本抽象开始。

1. 证券

代码清单6-2是对Instrument的抽象，也就是对证券交易所中买卖的证券进行建模。代码中首先定义Instrument的一般接口，然后针对Equity类型以及几种FixedIncome类型的证券进行特化。（如果需要了解各种证券类型的异同，请查阅4.3.2节的补充内容。）

代码清单6-2 Instrument的Scala模型

```
package api

import java.util.Date
import Util._

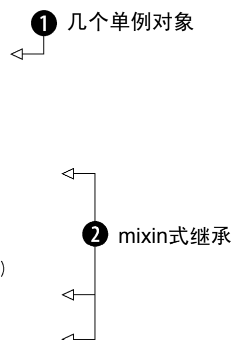
sealed abstract class Currency(code: String)
case object USD extends Currency("US Dollar")
case object JPY extends Currency("Japanese Yen")
case object HKD extends Currency("Hong Kong Dollar")

trait Instrument {
  val isin: String
}

case class Equity(isin: String, dateOfIssue: Date = TODAY)
  extends Instrument

trait FixedIncome extends Instrument {
  def dateOfIssue: Date
  def dateOfMaturity: Date
  def nominal: BigDecimal
}

case class CouponBond(
  override val isin: String,
  override val dateOfIssue: Date = TODAY,
  override val dateOfMaturity: Date,
  val nominal: BigDecimal,
  val paymentSchedule: Map[String, BigDecimal])
    extends FixedIncome
case class DiscountBond(
  override val isin: String,
  override val dateOfIssue: Date = TODAY,
  override val dateOfMaturity: Date,
  val nominal: BigDecimal,
  val percent: BigDecimal)
    extends FixedIncome
```



领域语汇在上面的实现中表达得很清晰，而且领域模型基本没有受到偶发复杂性的干扰。（关于偶发复杂性的详细讨论请参阅附录A。）这是本章建立的第一个领域模型，我们不妨在它身上多下一些功夫，看看哪些Scala特性对模型的表现力和简洁性有所贡献。

- 单例（singleton）对象，因为它只实例化一次的特点，此处被用于实现Currency类❶的几个特化实体。单例对象是Scala语言实现Singleton模式（参见6.10节文献[3]）的方式，弥补了Java语言中静态成员的所有不足。

- 在trait的组织下，通过继承❷来实现一种可扩展的对象层次关系。
- case类具有简化的构造函数调用形式。

我们还要再构建几个抽象才能开始编写DSL脚本。

2. 账户和交易

代码清单6-3是Account模型的Scala实现。客户与中介在Account这个领域实体上交易各种证券。

代码清单6-3　Account模型的Scala实现

```
package api

abstract class AccountType(name: String)
case object CLIENT extends AccountType("Client")
case object BROKER extends AccountType("Broker")

import Util._
import java.util.Date

abstract class Account(no: String, name: String, openDate: Date) {
  val accountType: AccountType

  private var closeDate: Date = _
  var creditLimit: BigDecimal = 100000          <-- 设置默认信用额度

  def close(date: Date) = {
    closeDate = date
  }
}

case class ClientAccount(no: String, name: String,
  openDate: Date = TODAY)
  extends Account(no, name, openDate) {
    val accountType = CLIENT
  }
case class BrokerAccount(no: String, name: String,
  openDate: Date = TODAY)
  extends Account(no, name, openDate) {
    val accountType = BROKER
  }
```

除了Account和Instrument模型，我们还需要一个代表证券交易本身的基本抽象。

代码清单6-4　Trade模型的Scala实现

```
package api

import java.util.Date

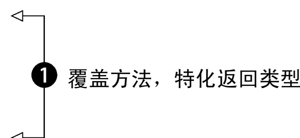
trait Trade {
  def tradingAccount: Account
  def instrument: Instrument
  def currency: Currency
  def tradeDate: Date
  def unitPrice: BigDecimal
```

```
  def quantity: BigDecimal
  def market: Market
  def principal = unitPrice * quantity

  var cashValue: BigDecimal = _
  var taxes: Map[TaxFee, BigDecimal] = _
}

trait FixedIncomeTrade extends Trade {
  override def instrument: FixedIncome
  var accruedInterest: BigDecimal = _
}

trait EquityTrade extends Trade {
  override def instrument: Equity
}
```



按照交易证券所属的类，我们定义了两种类型的交易。稍后你会了解，这两种类型的交易具有不同的特征，尤其是现金价值的计算方法很不一样。（关于交易的现金价值请参阅4.2.2节的补充内容。）代码清单6-4还有一点值得注意，我们覆盖了instrument方法❶，让它的返回类型正确地反映每一类交易所针对的证券类型。

我们仅仅为编写交易创造DSL而设置相关上下文就已经写了这么多代码，下一节该正式动笔了。顺便还要谈谈构建中用得上的Scala特性。

6.4 制作一种创建交易的 DSL

我一向认为应该先看到实物，再去研究它是怎么形成的。所以暂时别管具体怎么实现，先看看我们的交易DSL怎么创建新交易：

```
val fixedIncomeTrade =
  200 discount_bonds IBM
➥    for_client NOMURA on NYSE at 72.ccy(USD)

val equityTrade =
  200 equities GOOGLE
➥    for_client NOMURA on TKY at 10000.ccy(JPY)
```

第一项定义fixedIncomeTrade创建了一个FixedIncomeTrade的实例，为客户帐号NOMURA在纽约证券交易所（NYSE）按72美元的单价买入200张IBM的折价债券（DiscountBond）。

第二项定义equityTrade创建了一个EquityTrade的实例，为客户帐号NOMURA在东京证券交易所（TKY）按10 000日元的单价卖出200股Google的股票。


Scala知识点

- **隐含参数**（implicit parameter）。用户没有明确指定时，隐含参数由编译器自动提供。特别适用于设计精简的DSL语法。
- **隐式类型**转换是实现“限制了词法作用域的开放类”的秘诀。这种开放类近似于Ruby的猴子补丁，但比猴子补丁更好用。
- **命名参数和默认参数**用在Builder模式的实现当中，可以省略不少拖泥带水的代码。

如果不走DSL的路线，而是按照一般的API设计方式，通过某个具体类的构造函数来完成交易创建过程，那么代码差不多会是下面这样。代码清单6-5先给出`FixedIncomeTrade`的具体实现，然后演示了它的实例化过程。

代码清单6-5　`FixedIncomeTrade`的实现和实例化示例

```
package api

import java.util.Date
import Util._

case class FixedIncomeTradeImpl(
  val tradingAccount: Account,
  val instrument: FixedIncome,
  val currency: Currency,
  val tradeDate: Date = TODAY,
  val market: Market,
  val quantity: BigDecimal,
  val unitPrice: BigDecimal) extends FixedIncomeTrade

val t1 =
  FixedIncomeTradeImpl(
    tradingAccount = NOMURA,
    instrument = IBM,
    currency = USD,
    market = NYSE,
    quantity = 100,
    unitPrice = 42)
```

实现**`FixedIncomeTrade`** trait

实例化示例

DSL与一般API的差异明显。DSL版的创建过程更自然，更适合领域用户阅读；API版的编程味道比较浓，有许多语法细节需要注意，例如分隔参数项的逗号，实例化时要写出类名等。稍后你会发现，可读性强的DSL版为了实现操作的顺序执行，同样要满足许多约束条件。如果看重顺序的灵活性，可以选择Builder模式（参见6.10节文献[3]），但那样会带来Builder对象的可变性问题和方法链的收尾问题（参见6.10节文献[4]）。

本节开头介绍了DSL脚本的未来发展趋势，现在我们进入实现环节。

6.4.1　实现细节

请你再好好对比一下6.4节开头的DSL脚本和代码清单6-5中普通的构造函数调用写法。DSL脚本中几乎没有与领域语义无关的语法结构，这就是两者最明显的区别。前面说过，Scala允许省略表示方法调用的点运算符和方法参数使用的圆括号。就算在这样的有利条件之下，如果没有一种足够灵活的手段，还是不能把各种成分合理地结合起来，成为符合语言逻辑的脚本。那么，连接各语言成分的秘诀是什么？

1. 隐式转换

秘诀是Scala语言的`implicits`特性！我们以创建`FixedIncomeTrade`为例说明：

```
val fixedIncomeTrade =
  200 discount_bonds IBM for_client NOMURA on NYSE at 72.ccy(USD)
```

如果去掉各种语法糖，并且补上原来省略的点符号和圆括号，那么代码将变成下面这样的规范形式：

```
val fixedIncomeTrade =
  200.discount_bonds(IBM)
      .for_client(NOMURA)
      .on(NYSE)
      .at(72.ccy(USD))
```

当所有的方法调用和参数都披挂上它们应有的符号，看上去就和2.1.2节Java版指令处理DSL所用的Builder模式相差无几了。当前实现可以看做Builder模式的一种隐式实现。而我们也确实在实现中运用了Scala的隐式转换特性，令各部分以正确的次序组织起来，最终铺陈为有意义的DSL语句。

我们以`200 discount_bonds IBM`为例说明其原理。只要掌握这个词组的构建机制，例子的其他部分都不在话下，看看完整的代码，就知道每个零件的位置和作用。请看下面的代码片段：

```
type Quantity = Int
class InstrumentHelper(qty: Quantity) {
  def discount_bonds(db: DiscountBond) = (db, qty)
}

implicit def Int2InstrumentHelper(qty: Quantity) =
  new InstrumentHelper(qty)
```

我们定义的`InstrumentHelper`类接受一个`Int`作为输入，类中定义了`discount_bonds`方法。`discount_bonds`方法的参数是一个`DiscountBond`实例，返回由给定债券及其数量组成的一个`Tuple2`。接着我们定义了从`Int`到`InstrumentHelper`类的`implicit`转换。其作用自然是将输入的`Int`不动声色地转换为输出的`InstrumentHelper`实例，我们才得以在实例上调用`discount_bonds`方法。由于Scala允许省略点符号和圆括号，调用过程可以写成中缀形式，即`200 discount_bonds IBM`，这样看上去更自然。

只要用户定义好转换，Scala会在脚本的调用点插入必要的语义结构。脚本的其余部分也是同样的原理，在重重转换之际，构造`FixedIncomeTrade`实例所需的参数也收集到位，最终传入一个能生成`FixedIncomeTrade`实例的方法。隐式转换有一些需要掌握的惯用法，等看到完整代码时一并讲解。现在先看看图6-4，图中详细说明了完整的脚本执行过程。

为了真正理解图6-4，需要展示一个藏在API身后悄然发挥神秘作用的对象，详细分析它的全部代码。

2. 一连串的隐式转换

仔细观察图6-4，不难发现此起彼伏的隐式转换其实在默默扮演着builder的角色。这些转换行为逐渐拼合出最后的`FixedIncomeTrade`对象。代码清单6-6定义了执行各个转换的辅助函数。

代码清单6-6 `TradeImplicits`定义的转换函数

```
package dsl

import api._
object TradeImplicits {

  type Quantity = Int
```

```
    type WithInstrumentQuantity = (Instrument, Quantity)
    type WithAccountInstrumentQuantity =
      (Account, Instrument, Quantity)
    type WithMktAccountInstrumentQuantity =
      (Market, Account, Instrument, Quantity)
    type Money = (Int, Currency)

    class InstrumentHelper(qty: Quantity) {
      def discount_bonds(db: DiscountBond) = (db, qty)
    }

    class AccountHelper(wiq: WithInstrumentQuantity) {
      def for_client(ca: ClientAccount) = (ca, wiq._1, wiq._2)
    }

    class MarketHelper(waiq: WithAccountInstrumentQuantity) {
      def on(mk: Market) = (mk, waiq._1, waiq._2, waiq._3)
    }

    class RichInt(v: Int) {
      def ccy(c: Currency) = (v, c)
    }

    class PriceHelper(wmaiq: WithMktAccountInstrumentQuantity) {
      def at(c: Money) = (c, wmaiq._1, wmaiq._2, wmaiq._3, wmaiq._4)
    }
    //..
}
```

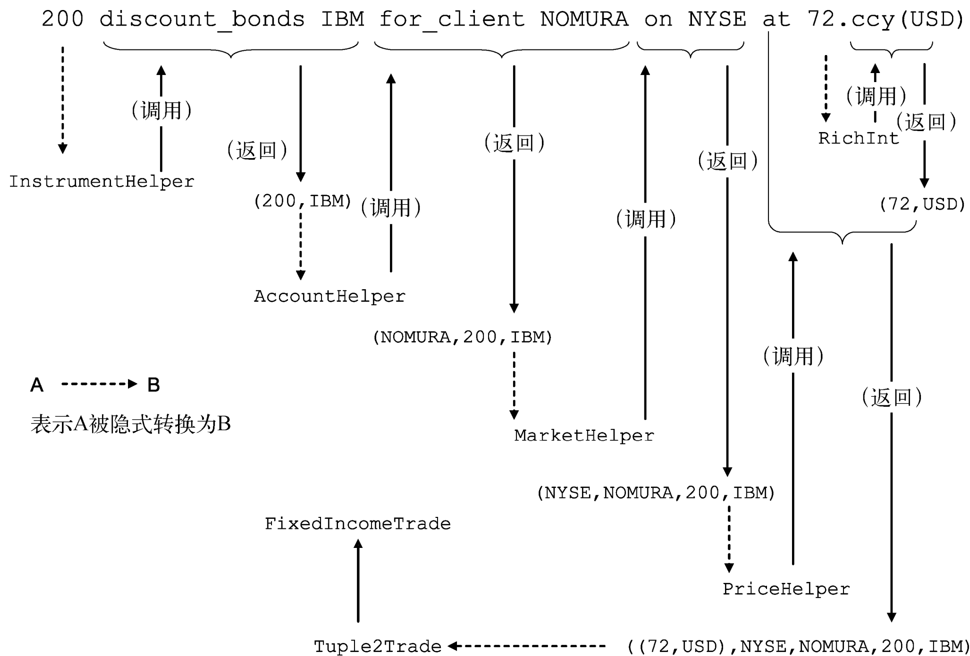


图6-4　一连串的隐式转换，最终构造出FixedIncomeTrade实例。从左至右阅读此图，跟随着箭头注意观察每一次隐式转换和创建辅助对象的子过程

代码清单6-7继续处理TradeImplicits对象，它把代码清单6-6列出的转换函数都定义成Scala的implicit。

代码清单6-7 TradeImplicits定义的implicits

```
object TradeImplicits {

  // 续代码清单6-6

  implicit def quantity2InstrumentHelper(qty: Quantity) =
    new InstrumentHelper(qty)
  implicit def withAccount(wiq: WithInstrumentQuantity) =
    new AccountHelper(wiq)
  implicit def withMarket(waiq: WithAccountInstrumentQuantity) =
    new MarketHelper(waiq)
  implicit def withPrice(wmaiq: WithMktAccountInstrumentQuantity) =
    new PriceHelper(wmaiq)
  implicit def int2RichInt(v: Int) = new RichInt(v)

  import Util._
  implicit def Tuple2Trade(
    t: (Money, Market, Account, Instrument, Quantity)) =
    {t match {
      case ((money, mkt, account, ins: DiscountBond, qty)) =>

      FixedIncomeTradeImpl(
        tradingAccount = account,
        instrument = ins,
        currency = money._2,
        tradeDate = TODAY,
        market = mkt,
        quantity = qty,
        unitPrice = money._1)
    }
  }
}
```


TradeImplicits对象属于dsl包，而所有的领域模型抽象都属于api包。这种看似不必要的划分有其深意。我们曾讨论过用基本领域模型作为底层基础，然后在上面建立DSL门面的设计惯例，想起来了吗？这个例子也按照同样的思路，把所有的领域模型抽象放入api包，语言层的抽象则放在dsl包里。另外，让这两个抽象层保持分离，这样有利于以后对同一个领域模型建立多种DSL。设计DSL时也应该始终遵循这种惯例。



利用Scala语言的implicits特性可以构造出开放类，类似于Ruby语言的猴子补丁和Groovy语言的ExpandoMetaClass。而且对于打开进行修改的类，Scala提供了控制其可见范围的方法。只要针对需要用到新增方法的局部词法作用域导入相应的模块，编译器会处理好其他事情。这种特性不会像Ruby的猴子补丁那样影响全局命名空间。

3. Implicits和词法作用域

我们利用implicits特性，通过将Int隐式转换为RichInt类，在Int身上添加了一个ccy方法。如果上述隐式转换在全局命名空间进行，则所有线程都能看这一修改。我们早在介绍Ruby

猴子补丁时就已经讨论过这样做的明显弊端。因此我们遵循一条黄金准则：`implicits`必须被限制在适当的作用域之内。也就是说，你应该划定隐式转换的词法作用域，然后在前面明确声明`import TradeImplicits._`，以免影响其他线程。

尽管隐式转换优点突出，可是使用它时不能直观地表现在代码的字面上，调试时又难以捉摸其来龙去脉。为此，Scala编译器特别提供了若干编译参数作为调试工具，方便你检查隐式转换（详见6.10节文献[2]）。

你刚刚完成平生第一段Scala DSL，它的表现力有没有给你惊艳的感觉？你一定要掌握本例实现的来龙去脉，否则请多看几遍图6-4，跟着箭头走，你对代码的理解也会越来越深入。

好了，现在你被Scala激起的兴奋心情也差不多该平复了。我们换个冷静的心态，考虑几个创建Scala DSL时需要注意的关键问题。

6.4.2　DSL实现模式的变化

仔细观察我们在领域模型上搭建的DSL层代码，可以辩认出一种模式的轮廓，再看看图6-4，这种模式显而易见。按照DSL脚本解析的方向，从左至右浏览示意图。我们连续运用Scala隐式转换，逐步构建一个*n*元组。6.4.1节提到，这样的构造方式其实就是Builder模式。只不过传统的实现方案会单独设立一个可变的builder抽象，负责构建实体，而此处采用了一种不可变的Builder模式变体。传统实现呈现一种命令式风格，builder对象连续发出方法调用，被一连串的方法调用所更新，同时每次方法调用都返回builder对象自身。相比之下，在这个不可变方案中，各个方法分属不同的类，要靠Scala的隐式转换把所有的调用粘合到一起。一次调用产生一个多元组，然后该多元组经过隐式转换，作为输入送到下一环节，生成下一个多元组。

你完全可以选择传统的实现方案。那样的话，会有什么不同吗？传统Builder模式的方法调用序列写起来十分灵活方便，但问题是用户最后必须调用一个收尾方法来结束构建过程（参见6.10节文献[4]）。相比之下，本例当前的实现方式已经将调用序列固定在DSL里面，如果用户没构造完交易对象就提早终止调用序列，那么编译器会发出提醒。两种方案并没有绝对的优劣之分，无非是一个设计决策。

传统的Builder模式有一个可变的builder对象，用户通过它的连贯接口发起链式的方法调用，完成对该builder对象的修改。本例中的Builder模式实现由一系列隐式转换构成，每一步转换产生的对象都是不可变的。尽量采用不可变性的抽象设计是一种好习惯。

在领域抽象上面建立DSL门面离不开几项Scala特性，表6-3总结了这些特性。

表6-3　交易创建DSL用到的Scala特性

Scala语言特性	作　　用
灵活的语法。由于省略点符号（.）和圆括号，形成了中缀表示法	使DSL更容易阅读，表达更清晰
隐式转换	通过限制了词法作用域的开放类，可以向`Int`等内建类加入新方法对象链接
命名参数和默认参数	使DSL更容易阅读

创建交易对象的DSL至此告一段落。下一节要为更多的业务规则建立DSL，而且DSL写成的每一条规则都能让领域专家看懂并把关。基于DSL的开发，其出发点正是促进与领域专家的沟通，协助专家查验由开发者实现的业务规则。下一步的DSL开发需要我们再准备一些领域抽象作为底层的实现模型。

6.5　用 DSL 建模业务规则

业务规则是DSL的一个应用热点。业务规则属于领域模型中可配置的部分，正是最需要领域专家过目的环节。如果DSL非常容易上手，连领域专家都能（像老鲍那样）写上几行测试，那简直是锦上添花的好事。我们的DSL准备针对交易的税费计算进行建模，图6-4说明了计算的详情。

表6-4　DSL将要建模的业务规则：计算交易的税费

步　骤	说　明
(1) 执行交易	买卖双方在证券交易所产生交易
(2) 计算税费	已发生的交易需要计算相应的税费。计算逻辑由交易类型、交易的证券、进行交易的证券交易所等因素决定
	买卖双方按照交易净值进行结算，税费是交易净值的核心组成部分

我们的DSL要求能被领域专家老鲍看明白，理解其中的业务规则，然后检查规则的完备性，验明规则的正确性。那么，第一步应该做什么呢？还用问吗！当然是建立税费的领域抽象，不然DSL层就成空中楼阁了。

不过，聪明的读者肯定急着想看下一轮的DSL实现，没耐心再听我介绍一遍领域建模。为了提起你的兴趣，不如我们打个岔，先在手头已有领域模型的基础上，构建一个实现业务规则的DSL。这个小练习除了能提神，还能演示Scala语言一项重要的函数式特性，它应用广泛，并能大大改善一种最常用的面向对象设计模式。



Scala知识点

- **模式匹配**可用于实现函数式的抽象，还可实现一种可扩展的Vistor模式。
- **高阶函数**是Scala语言代表性的函数式编程特性。可用于实现组合子。
- **抽象val**和**抽象类型**用于设计开放的抽象。开放抽象可适时组合为具体的抽象。
- **自类型标注**（self-type annotation）可以用来建立抽象间的关联。
- **偏函数**（partial function）是可对其定义域的一个子集进行求值的表达式。

6.5.1　模式匹配如同可扩展的Visitor模式

Scala语言的case类除了构造函数的调用语法较为简单，还可以对析构对象进行模式匹配（Scala模式匹配的用法详见6.10节文献[2]）。Haskell等函数式语言的代数数据类型（algebraic data

types，详见http://en.wikipedia.org/wiki/Algebraic_data_type）一般都要用到模式匹配特性。对case类使用模式匹配，是为了实现一种通用的、可扩展的Visitor模式（参见6.10节文献[3]）。

Visitor模式用于我们的DSL设计，可以改善领域规则的表达，使规则看上去更清晰和直观。模式匹配这种函数式实现范式，配合case类的用法，能大大提高DSL的表达能力和扩展能力，并且不会像一般面向对象的Visitor实现那样，容易出现领域规则被深埋在对象层次里面的问题。与传统的面向对象Visitor实现相比，模式匹配结合Scala中的case类能够打造出更多可扩展的解决方案，欲了解这方面的更多详细信息，参加6.10节文献[5]。

我们考虑为这样一条业务规则建立DSL：对于在今天之前开户的所有账户，将其额度提高10%。

领域模型沿用代码清单6-3的`Account`抽象及其具体实现`ClientAccount`和`BrokerAccount`。（客户账户的讨论见3.2.2节的补充内容。中介账户是中介方在证券交易组织开设的账户。）实现上述规则的要点，一是从系统内的所有帐户中找出客户帐户，二是从客户帐户中找出需要修改额度的帐户。具体实现请看下面的`raiseCreditLimits`函数。

```
def raiseCreditLimits(accounts: List[Account]) {
  accounts foreach {acc =>                                         ❶ 模式匹配
    acc match {
      case ClientAccount(_, _, openDate) if (openDate before TODAY) =>
        acc.creditLimit = acc.creditLimit * 1.1
      case _ =>
    }
  }
}
```

业务规则被表达为对case类进行模式匹配的规则，请注意体会这种写法直观、清晰的特点。编译器会在后台将case语句❶展开为偏函数，偏函数的定义范围只限于case子句中指定的取值。我们的领域规则只针对客户帐户，所以就在第一条case子句里面把定义范围限定为`ClientAccount`实例——用模式匹配来建模领域规则就是这么轻松。第二条case子句是“不关心”子句，里面的下划线符号（_）代表了我们不关心的其他类型账户。关于模式匹配和偏函数这两项Scala语言特性的详细介绍，请查阅6.10节文献[2]。

这段代码能算DSL吗？算。它对领域规则的表述，达到了领域专家能理解的直观程度。它把实现浓缩在很短的篇幅里，领域专家不需要来回翻查代码就能掌握规则语句的语义。最后，它的表述只落墨在规则中明确提及、有重要意义的属性上，其他无关紧要的部分都用一句“不关心”一带而过。

DSL的表达能力只要满足用户需要即可

DSL不一定非要向自然语言靠拢。我重申：对DSL表达能力的要求是，足够满足用户需要。本例的代码片段由程序员使用，所以只要把规则的意图表达清楚，让程序员能维护、领域用户能看懂，这样就足够了。

上面的DSL片段应该能让你初步了解业务规则建模，接下来我们要继续完成本节开头搁置的任务——建立税费的领域模型。下一节有许多新的Scala建模技巧在等着你，绝不会辜负你的学习

热情。所以，快来杯咖啡提提神，马上要开始了。

6.5.2 充实领域模型

问题域的几个基本抽象，Trade、Account和Instrument都已经准备就绪，可以开始考虑税费组件的设计了。计算交易的现金价值这项功能，需要若干税费计算方面的组件与Trade组件共同完成。

税费计算的职责应该设立一个单独的抽象去承担。税费的计算方法是模型中必不可少的业务规则，它会因为业务所处的国家、交易所而变化。根据前面的学习，想必你已经总结出规律：凡是会变化的业务规则，DSL可以让规则的表述直白、清晰、易于维护，进而减轻你的工作负担。

图6-5是我们的解决方案的总体组件模型，说明了税费抽象与Trade组件的交互情况。

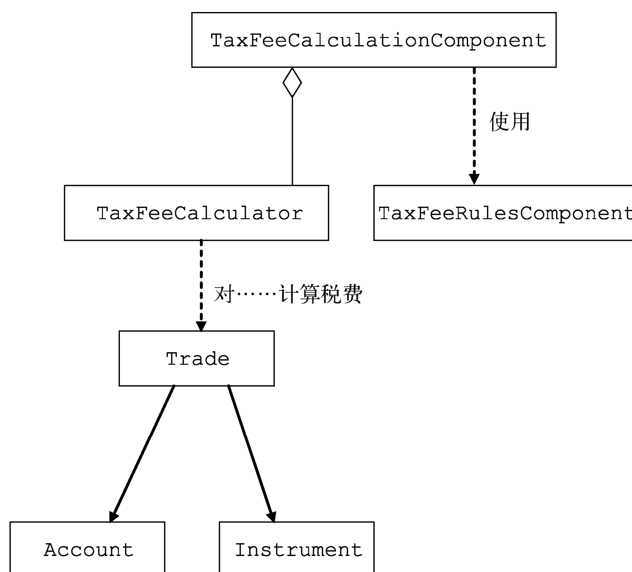


图6-5　交易领域模型中的税费组件。此类图反映了TaxFeeCalculationComponent与其他协作抽象之间的静态关系

除了Account，图6-5中列出的抽象都被建模为Scala trait的形式。因此它们可以灵活地关联在一起，并在运行时与合适的实现组合在一起，构成恰当的具体对象。请看代码清单6-8中TaxFee-Calculator和TaxFeeCalculationComponent的实现代码。

代码清单6-8　税费计算组件的Scala实现

```
package api

sealed abstract class TaxFee(id: String)
case object TRADE_TAX extends TaxFee("Trade Tax")
case object COMMISSION extends TaxFee("Commission")
```

❶ 各单例对象

```
case object SURCHARGE extends TaxFee("Surcharge")
case object VAT extends TaxFee("VAT")

trait TaxFeeCalculator {
  def calculateTaxFee(trade: Trade): Map[TaxFee, BigDecimal]
}

trait TaxFeeCalculationComponent { this: TaxFeeRulesComponent =>
  val taxFeeCalculator: TaxFeeCalculator

  class TaxFeeCalculatorImpl extends TaxFeeCalculator {
    def calculateTaxFee(trade: Trade): Map[TaxFee, BigDecimal] = {
      import taxFeeRules._
      val taxfees =
        forTrade(trade) map {taxfee =>
          (taxfee, calculatedAs(trade)(taxfee))
        }
      Map(taxfees: _*)
    }
  }
}
```

❷ 计算给定交易的税费

❸ 自类型标注

❹ 抽象val

❺ 对象导入语法

深入分析这段代码可以帮助你理解整个组件模型是怎样联系起来的。请看表6-5的详细分析。

表6-5　剖析税费计算组件的Scala DSL实现模型

抽　　象	在DSL实现中的作用
TaxFee	TaxFee抽象是值对象（参见6.10节文献[6]），代表一个税费品种。不同的税费品种用不同的Scala单例对象❶来表示 注意：作为值对象，各税费类型对象不可变
TaxFeeCalculator	TaxFeeCalculator抽象负责计算给定交易应承担的全部税费❷
TaxFeeCalculationComponent	它是联系起整组税费相关抽象的中心，其他抽象围绕着它形成交易税费的计算核心 TaxFeeCalculationComponent通过自类型标注的与TaxFeeRulesComponent协作❸，通过抽象val与TaxFeeCalculator协作❹ 本设计的优点： ■ 抽象与实现解耦。你可以为TaxFeeCalculationComponent的两个协作抽象提供任意实现 ■ 实现可推迟到创建TaxFeeCalculationComponent的具体实例时

Scala语言的自类型标注

自类型标注可赋予组件内的自指对象this额外的类型。这种用法含蓄地声明trait TaxFeeCalculationComponent extends TaxFeeRulesComponent。

只不过我们并不立即创建这条编译时依赖关系，而是用自类型标注的方式承诺，在具体TaxFeeCalculationComponent对象实例化时，一定会混入TaxFeeRulesComponent。代码清单6-10以及后续创建具体对象的代码清单6-11和代码清单6-12履行了上述承诺。

注意，在TaxFeeCalculatorImpl#calculateTaxFee内部有一处针对taxFeeRules的

导入❺，taxFeeRules也是TaxFeeRulesComponent内的抽象val。

TaxFeeRulesComponent被声明为自类型标注，即向Scala编译器宣告它是this的有效类型之一。自类型标注的原理详情，请参阅6.10节文献[2]。

寥寥几行代码就已经串起多个组件。省代码是Scala带来的好处，也是运用高级抽象编程的结果。下一节将完成TaxFeeRulesComponent实现，并且设计一种DSL来定义税费计算的领域规则。

6.5.3　用DSL表达税费计算的业务规则

我们首先搭建规则组件的领域模型，它是一个trait，对外公开税费计算方面的主要契约。为简单起见，假设了一种简化的情况，现实中的规则要复杂烦琐得多。

```
package api

trait TaxFeeRules {
  def forTrade(trade: Trade): List[TaxFee]
  def calculatedAs(trade: Trade): PartialFunction[TaxFee, BigDecimal]
}
```

❶ 适用于给定交易的**TaxFee**

❷ 具体的计算方法

第一个方法forTrade❶返回给定交易应缴纳的税费品种列表。第二个方法calculatedAs❷针对给定交易计算具体某一项税费。

现在看看TaxFeeRulesComponent组件，它除了建立税费计算DSL，还提供了TaxFeeRules的一个具体实现。该组件如代码清单6-9所示。

代码清单6-9　表达税费计算业务规则的DSL

```
package api

trait TaxFeeRulesComponent {
  val taxFeeRules: TaxFeeRules

  class TaxFeeRulesImpl extends TaxFeeRules {
    override def forTrade(trade: Trade): List[TaxFee] = {
      (forHKG orElse
         forSGP orElse
           forAll)(trade.market)
    }

    val forHKG: PartialFunction[Market, List[TaxFee]] = {
      case HKG =>
        List(TradeTax, Commission, Surcharge)
    }

    val forSGP: PartialFunction[Market, List[TaxFee]] = {
      case SGP =>
        List(TradeTax, Commission, Surcharge, VAT)
    }

    val forAll: PartialFunction[Market, List[TaxFee]] = {
      case _ => List(TradeTax, Commission)
```

❶ 用组合子把几个**TaxFee**列表连接起来

❷ 针对中国香港市场的专门列表

❸ 针对新加坡市场的专门列表

❹ 针对其他国家/地区的通用列表

```
    }

    import TaxFeeImplicits._
    override def calculatedAs(trade: Trade):
        PartialFunction[TaxFee, BigDecimal] = {
      case TradeTax   => 5.  percent_of trade.principal
      case Commission => 20. percent_of trade.principal
      case Surcharge  => 7.  percent_of trade.principal
      case VAT        => 7.  percent_of trade.principal
    }
  }
}
```

❺ 税费计算的领域规则

TaxFeeRulesComponent是对TaxFeeRules抽象的发展，而且提供了TaxFeeRules的实现，当然你也可以自己提供一个实现来代替它。TaxFeeRulesComponent仍然是一个抽象组件，因为里面的taxFeeRules只有抽象的声明。最后组装各部分组件时才提供所有的具体实现，构建出具体的TradingService。现在先来仔细研究这段实现代码，看看DSL怎么判断应缴税费品种，又怎么算出税费金额。

1. 选出合适的应缴税费品种列表

请看TaxFeeRulesImpl里面的DSL实现。forTrade方法只有一行，它是用Scala组合子和函数式风格组织起来的。附录A将介绍，组合子是高阶函数的优秀组织手段。（不读附录A，你可错过了好东西。）

组合子有让DSL语言精炼的效果。它是函数式编程最有吸引力的部分之一。既然Scala给了你函数式编程的强大工具，该用组合子组织语言时就别犹豫。为给定交易找到适当税费集合的业务规则，用自然语言描述就是下面这样：

> “为在中国香港市场进行的交易提供专门的列表，或者为在新加坡市场进行的交易提供专门的列表，或者为在其他市场进行的交易提供通用列表。”

Scala语言的偏函数

偏函数**只是**为其参数的部分取值而定义的。Scala的偏函数形式上是由一组模式匹配case语句构成的代码块。请看下面的例子：

```
val onlyTrue: PartialFunction[Boolean, Int] = {
  case true => 100
}
```

onlyTrue是一个PartialFunction。它只对其定义域（全体Boolean值）的一部分，即取值为true的情况做了定义。PartialFunction trait内含isDefinedAt方法，可以判断某个领域值是否属于PartialFunction的定义范围。例子如下所示：

```
scala> onlyTrue isDefinedAt(true)
res1: Boolean = true
scala> onlyTrue isDefinedAt(false)
res2: Boolean = false
```

现在对比着以上规则读一下`forTrade`❶里面那一句实现。你会发现代码中的规则表达严丝合缝地对应了上面自然的叙述形式，而且浓缩在一个非常紧凑的API界面上。代码中用到了`orElse`组合子，它的作用是连接多个Scala偏函数，然后选取第一个对给定参数取值有定义的偏函数。

按照代码清单6-9的定义，`forTrade`方法只有在`market`不是中国香港，也不是新加坡时，才返回一个通用的`TaxFee`对象列表。理解了`forTrade`的原理，掌握了Scala偏函数的组合方法，也就知道了`forHKG`❷、`forSGP`❸、`forAll`❹这几个高阶函数的工作原理。

2. 计算税费

现在该说具体的税费计算了，这是DSL要解决的第二部分业务规则。请看代码清单6-9的`calculatedAs`❺方法。你能看出它实现了什么规则吗？

`calculatedAs`方法把领域规则表达得十分清楚，这又是Scala模式匹配的功劳。它的几条`case`子句都经过`implicits`的妙手润色。通过`implicits`手法给`Double`类增加`percent_of`方法，然后写成中缀形式，就得到代码清单6-9中的结果。隐式转换使用前需要先将其定义导入当前作用域，也就是下面的`TaxFeeImplicits`对象：

```
package api

object TaxFeeImplicits {
  class TaxHelper(factor: Double) {
    def percent_of(c: BigDecimal) = factor * c.doubleValue / 100
  }

  implicit def Double2TaxHelper(d: Double) = new TaxHelper(d)
}
```

导入`TaxFeeImplicits`对象之后，我们就可以像`calculatedAs`方法那样，把句子写成符合领域语法的形式，业务专家们看了一定会高兴的。

3. DSL和API有什么区别

6.5节主要介绍了两件事，一是学习在底层实现模型上搭建创建领域实体的DSL脚本，其次学习为业务规则构建DSL。Scala提供了几种手段，可在面向对象的领域模型上实现表意清晰的API。这些手段前面已经介绍过。我在两部分的实现中都多走了一步，运用Scala的隐式转换提供的开放类去改善语言的表达效果。但即使不利用这种贴心的`implicits`特性，只要综合运用面向对象和函数式两方面的特性，已经足够为领域模型建立一套表达力充分的API。

既然如此，你肯定会问：到底内部DSL和API有什么区别呢？坦白讲，区别不大。如果一套API具有充分的表达能力，能向用户清楚揭示领域语义，同时又不增加额外的非本质复杂性，那么它就可以算作一种内部DSL。纵观书中被挂上DSL名号的代码片段，我总是根据它们面向领域用户的表现力来决定它们的称呼。DSL实现者需要维护代码，领域专家需要理解语义；要想不多做语法上的包装同时满足这两方面的需要，你选择的实现语言必须拥有建立并组合高阶抽象的能力。我建议你再次重温附录A中树立的抽象设计原则。

前面提到，图6-5列出的组件都是抽象的，我们有意把组件设计成trait。不过你还没见识过trait的真正实力，把抽象的trait组合成可实例化的具体领域实体时，你才知道这种组合威力有多大。

下一节会将各种交易组件组合起来，构建一些具体的交易服务，然后以交易服务为根基建立DSL的语言抽象。

6.6　把组件装配起来

带着为税费计算业务规则建立DSL的经验，我们来准备下一道DSL大菜，这道菜的材料还欠几味抽象。



Scala知识点

- Scala的**模块**，即`object`关键字。允许通过组合抽象的组件来定义具体的实例。
- 各种**组合子**，如`map`、`foldLeft`、`foldRight`。

本节将介绍怎样运用Scala的mixin式继承来组合不同的trait。你还会看到Scala语言支持的另一种抽象形式——基于类型的抽象。可供选择的抽象组合手段越多，领域模型的可塑性就越强，也越容易衍生出合适的DSL语法。

6.6.1　用trait和类型组合出更多的抽象

领域模型里面有一部分抽象扮演对外的窗口，直接面向最终用户，领域服务即为其中之一。领域服务使用各种实体和值对象（详见6.10节文献[6]），向用户履行契约。在代码清单6-10中，`TradingService`是一个典型的领域服务，但比真实的使用案例简化很多。

代码清单6-10　服务基类`TradingService`的Scala实现

```
package api

trait TradingService
  extends TaxFeeCalculationComponent
  with TaxFeeRulesComponent {
  type T <: Trade

  def taxes(trade: T) =
    taxFeeCalculator.calculateTaxFee(trade)

  def totalTaxFee(trade: T): BigDecimal = {
    taxes(trade).foldLeft(BigDecimal(0))(_ + _._2)
  }

  def cashValue(trade: T): BigDecimal
}
```

❶ 利用trait完成的mixin式继承
❷ 抽象类型
❸ 基于组合子的编程方式
❹ 抽象方法

图6-6简要讲解了服务契约的履行过程，同时指出其中用到的一些Scala特性。

到目前为止我们还没有具体化任何抽象，几个trait都还带着抽象类型，下一节将给出定义。Scala语言提供了异常丰富的抽象设计手段供你选择。设计时，应该针对手头的问题，挑选最合适的工具，也别忘了参照附录A讨论的设计原则。

表6-6　剖析代码清单6-10中的**TradingService**领域服务

特　性	说　明
mixin与现有抽象组合在一起的强大能力	TradingService被混入了两个组件，TaxFeeCalculationComponent和TaxFeeRulesComponent❶ **注意：** 在mixin方式下，我们不但继承了接口，还继承了可选的实现。mixin才是正确的多重继承机制
针对交易类型的抽象	TradingService对交易类型做了抽象❷。这样的安排很好理解，因为不同类型的交易需要区别对待，由各自专门的交易服务去处理。虽然服务基类TradingService不理会具体的交易类型，但交易必须满足从属于Trade基类这个基本条件 **什么时候具体化类型T** 到了具体化TradingService时，我们会为抽象交易类型T提供一个具体实现
税费计算的核心逻辑位于totalTaxFee方法	服务中定义了具体方法totalTaxFee❸，用于合计组件中算出的税费项目，计算通过foldLeft组合子完成。Scala组合子foldLeft的原理以及占位符“_”的用法详见附录D **提示：** 应该优先使用组合子，其次才考虑递归或迭代
通过抽象方法将实现工作推给子类	cashValue是抽象方法❹，因为具体的逻辑与服务要处理的交易类型有关，所以把它留给子类型去实现

6.6.2　使领域组件具体化

EquityTradingService为股票交易提供交易服务。它是一个具体的组件，只需针对它生成的服务实例化一次。代码清单6-11用Scala的单例对象表示法（具体参见6.10节文献[2]）来建模EquityTradingService。

代码清单6-11　针对股票交易的交易服务具体实现

```
package api

object EquityTradingService
  extends TradingService {

  type T = EquityTrade                                    ❶ 提供具体的类型

  val taxFeeCalculator = new TaxFeeCalculatorImpl         ❷ 提供具体的val
  val taxFeeRules = new TaxFeeRulesImpl

  override def cashValue(trade: T): BigDecimal = {
    trade.principal + totalTaxFee(trade)                  ❸ 提供具体的方法
  }
}
```

看上去挺简单的，对吧？写法有以下几个要点：

- 用一个具体类型EquityTrade❶代替基类中定义的抽象交易类型；
- 先前混入到基类的trait还留下几个抽象的val，我们要一一提供具体的实现❷；
- 针对股票交易的cashValue，给出具体的计算方法❸。

参照EquityTradingService的实现方式，我们继续实现固定收益型交易FixedIncomeTrade对应的具体交易服务FixedIncomeTradingService，如代码清单6-12所示。

代码清单6-12　Scala中针对固定收益型交易的交易服务

```
package api

object FixedIncomeTradingService
  extends TradingService with AccruedInterestCalculationComponent {    // ❶ 添加计算应计利息的mixin

  type T = FixedIncomeTrade

  val taxFeeCalculator = new TaxFeeCalculatorImpl
  val accruedInterestCalculator = new AccruedInterestCalculatorImpl    // 应计利息计算器的具体实例
  val taxFeeRules = new TaxFeeRulesImpl

  def accruedInterest(trade: T): BigDecimal = {
    accruedInterestCalculator.calculateAccruedInterest(trade)
  }

  override def cashValue(trade: T): BigDecimal = {
    trade.principal +
      accruedInterest(trade) + totalTaxFee(trade)
  }
}
```

注意，我们在核心抽象上额外混入了AccruedInterestCalculationComponent组件❶，它的功能是计算“应计利息”。固定收益型证券一般都含有应计利息，而且固定收益型交易的现金价值应该将此利息计算在内。这条业务规则不难从FixedIncomeTradingService的定义中看出来。

本节先定义了领域服务抽象，然后将前面几节构建的组件装配上去，构造出可以在DSL中直接使用的具体Scala模块。



这个练习展示了Scala的真正威力，它可以推迟到最后时刻才进行具体的实现。Scala的这种能力源自**抽象值**、**抽象类型**、**自类型标注**这三大支柱的支撑。除此之外，**mixin式继承**灵活的抽象组合能力也起了很大作用。Scala给了你丰富的手段去设计可扩展的各式组件。

我们在基础领域模型组件的基础上构建了一套DSL。就Scala而言，我把DSL层看做根据用户需求逐渐演化的一组抽象。按照这样的思路，在对交易系统中的不同用例进行建模之后，你将得到一个多层次的组合抽象模型。当市场规则有变，需要在现有抽象上加入新规则时，也就是当现有DSL需要与新DSL携手合作时，应该怎样实施？下一节说说怎么用Scala的类型系统来做这件事。

6.7　组合多种 DSL

能表明不同意图的各种抽象聚在一起，构成应用的领域模型。DSL层作为领域模型之上的一层门面，只有它的抽象级别正确时，才表现出功用和可扩展性。本节把DSL整体作为一个抽象单

元，讨论不同DSL之间的组合方法。以后遇到需要按不同方式将多个Scala DSL组合在一起的情况，将会用到这方面的技巧。作为讨论用的案例，我们从交易系统领域内挑选了市场规则DSL和交易处理的核心业务规则作为集成对象。

设计好DSL的抽象之后可以通过Scala的子类型化手段来扩展它。子类型化会产生一个层级关系的关联结构，针对相同的核心语言，有各种特化抽象分别提供不同的实现。这不就是所谓的多态吗？没错，6.7.1节将利用DSL的多态关系来组合它们。6.7.2节则讨论怎样组合那些没有亲缘关系的DSL。毕竟不同的DSL一般有着各自独立的发展轨迹，DSL的发展往往也不受应用生命周期的约束。应用架构必须提供良好的环境，让各式各样的DSL结构能无缝地组合起来。

6.7.1 扩展关系的组合方式

交易输入系统之后，将经过一系列常规的交易处理流程，其中第一个步骤是*交易充实*。该步骤向交易记录补充一些上游系统没有直接提供的衍生信息。信息包括交易的现金价值、应缴税费等，不同类型的交易证券还会有其他各式各样的信息。

1. DSL的扩展

下面的脚本骨架看上去还不是DSL应该有的样子，我们会在后续的讨论中为它添加血肉。为交易处理流程定义方法时，还会用到之前实现的一些组件。

```
package dsl

trait TradeDsl {
  type T <: Trade
  def enrich: PartialFunction[T, T]        ❶ 抽象方法
}
```

我们的专用语言现在的语义还很单薄，它仅仅定义了一个`enrich`方法，用于为输入系统的交易充实信息❶。

现在分别为`FixedIncomeTrade`和`EquityTrade`两种交易定义具体的`TradeDsl`实现，如代码清单6-13和代码清单6-14所示。`FixedIncomeTrade`对应的DSL要用到我们之前设计的`FixedIncomeTradingService`抽象。

代码清单6-13 针对`FixedIncomeTrade`的交易DSL

```
package dsl

import api._
trait FixedIncomeTradeDsl extends TradeDsl {
  type T = FixedIncomeTrade

  import FixedIncomeTradingService._

  override def enrich: PartialFunction[T, T] = {
    case t =>
      t.cashValue = cashValue(t)
      t.taxes = taxes(t)                          固定收益交易的应计利息
      t.accruedInterest = accruedInterest(t)
```

```
      t
    }
}

object FixedIncomeTradeDsl extends FixedIncomeTradeDsl
```

DSL的具体实例

EquityTrade对应的DSL要用到EquityTradingService抽象。

代码清单6-14　针对EquityTrade的交易DSL

```
package dsl

import api._
trait EquityTradeDsl extends TradeDsl {
  type T = EquityTrade

  import EquityTradingService._

  override def enrich: PartialFunction[T, T] = {
    case t =>
      t.cashValue = cashValue(t)
      t.taxes = taxes(t)
      t
  }
}

object EquityTradeDsl extends EquityTradeDsl
```

代码清单6-13的FixedIncomeTradeDsl和代码清单6-14的EquityTradeDsl，为同样的核心语言TradeDsl分别提供具体的语言实现。它们的交易充实语义，分别用6.6节的两个TradingService具体类来实现。从图6-6的类图可以看出这两个语言抽象之间的关系。

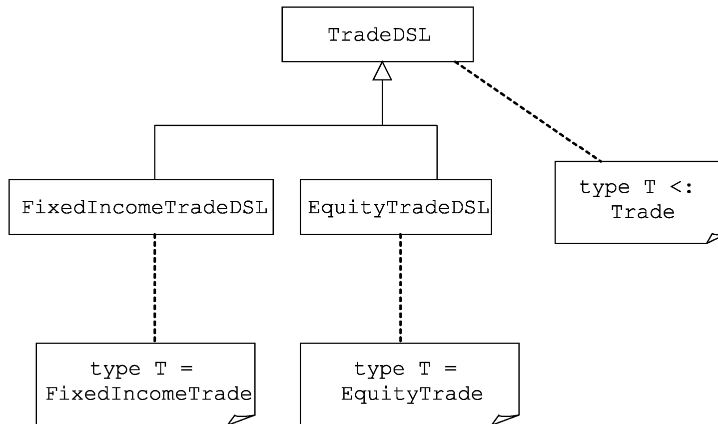


图6-6　TradeDSL有一个抽象类型成员T <: Trade，但EquityTradeDSL在相应的位置上指定了具体的类型T = EquityTrade，FixedIncomeTradeDSL指定了具体类型T = FixedIncomeTrade。EquityTradeDSL和FixedIncomeTradeDSL是TradeDSL的特化抽象

因为FixedIncomeTradeDSL和EquityTradeDSL扩展自同一个父抽象，平常那些围绕继承

关系的多态用法都可以套用上去。但如果我们需要定义这么一个TradeDSL子类型，它并不针对特定的交易类型，同时它所建模的业务规则需要结合EquityTradeDSL和FixedIncome-TradeDSL的语义，又应该怎么做呢？下一节的例子演示了Scala的另一种组合手段。

2. 通过可插拔替换的语义来组合

业务规则会随市场条件、监管规则，还有其他各种因素而改变。假设券商组织为了促进高价值的交易，宣布了这样一条新的市场规则：

> “对于任何在纽约证券交易所进行的交易，如果基本价值>1000 USD，在计算其净现金价值时，应对基本价值进行10%的折扣。”

现在不管交易的类型是EquityTrade还是FixedIncomeTrade，都需要在交易充实算法中实现以上规则。这条规则不应该成为核心现金价值计算的一部分，毕竟让短期促销用的市场规则影响系统的核心逻辑是不合理的。类似的领域规则，我们希望实现成洋葱一样的层叠结构，既可灵活地增减，又不影响核心抽象（请比照装饰器模式来理解）。代码清单6-15对TradeDsl的语义进行扩充，以反映上述针对个别市场的新规则。

代码清单6-15　为TradeDsl扩充新语义——新增业务规则

```
package dsl
import api._

trait MarketRuleDsl extends TradeDsl {
  val semantics: TradeDsl
  type T = semantics.T

  override def enrich: PartialFunction[T, T] = {
    case t =>
      val tr = semantics.enrich(t)
      tr match {
        case x if x.market == NYSE && x.principal > 1000 =>
          tr.cashValue = tr.cashValue - tr.principal * 0.1
          tr
        case x => x
      }
  }
}
```

❶ 被嵌入的内层语义

调用内层语义

给内层DSL披挂上附加规则

这段代码是讨论DSL组合的一个小高潮。注意semantics抽象val的作用，内层DSL被嵌入到这个位置，等待与新领域规则进行组合❶。内部DSL的另一种叫法正好是“内嵌式DSL”。大多数时候，DSL的语义是和实现硬性绑定在一起的。但这个例子的情况不一样，我们希望待组合DSL的语义是可以插拔替换的。那样的话，就能在被组合的各DSL之间保持松散的耦合度。除此之外，运行时的插拔机制有利于推迟确定具体的实现。代码清单6-16为EquityTradeDsl、FixedIncomeTradeDsl分别与新规则MarketRuleDsl的组合体定义具体对象。

代码清单6-16　组合DSL

```
package dsl
```

```
object EquityTradeMarketRuleDsl extends MarketRuleDsl {
  val semantics = EquityTradeDsl
}

object FixedIncomeTradeMarketRuleDsl extends MarketRuleDsl {
  val semantics = FixedIncomeTradeDsl
}
```

稍后你会看到各种DSL组合成果的汇总。在此之前，我们要用前面学过的函数式组合子知识，再扩充一下`TradeDsl`的功能。组合子在函数式层面和对象层面都可以表现出它的组合性语义。

3. 通过函数式组合子来组合

本节开头曾经提到交易的处理流程，还记得吗？在执行交易充实步骤之前，首先要验证交易的有效性。而在充实之后，交易还要被登记到会记系统的账簿之中。现实中的交易处理过程其实不止这几步，但作为演示，我们姑且让例子简单一些。应该怎样用Scala DSL建模这个流程序列呢？

组成流程序列的步骤适合用`PartialFunction`组合子来建模，而模式匹配可以把规则表达得清晰直白。代码清单6-17把原先的`TradeDsl`骨架变得丰满了一些，增加了一套控制结构来表示对流程步骤的规定。

代码清单6-17　在DSL中建模交易的处理流程

```
package dsl

import api._

trait TradeDsl {
  type T <: Trade

  def withTrade(trade: T)(op: T => Unit): T = {      ❶ 加入自定义操作

    if (validate(trade))
      (enrich andThen journalize andThen op)(trade)   ❷ 基于组合子的中缀运算
    trade
  }

  def validate(trade: T): Boolean = //..
  def enrich: PartialFunction[T, T]
  def journalize: PartialFunction[T, T] = {
    case t => //..
  }
}
```

为什么要把`enrich`定义成`PartialFunction`呢？Scala的偏函数具有令人赞叹的组合能力，特别适合用来构建高阶结构。

`withTrade`被定义成一个控制结构，交给它一笔交易，它会从头到尾执行完交易处理的全部流程。这个控制结构还具有向流程中插入自定义操作的能力，自定义操作通过`(op: T => Unit)`参数❶传入`withTrade`。传入的参数应该是一个作用于交易，且没有返回值的函数。日志、发送邮件、审计跟踪等函数就具备这样的特点，有副作用，但不影响操作后的返回值。类似的具有副作用的操作很适合通过这个途径插入到交易处理流程。

`withTrade`代码中的模式匹配块值得一说，它只用四行代码就将全部领域规则囊括在内。另外`andThen`组合子❷也出色地表达了各步骤的既定顺序。

4. 使用组合完毕的DSL

DSL组合完毕后的实际表现请看代码清单6-18。这段脚本用我们的“交易创建DSL”建立一笔交易，然后执行充实、验证等各个步骤，完成交易处理的全过程，最后联合市场规则DSL算出交易最后的现金价值。

代码清单6-18 交易处理流程DSL

```
import FixedIncomeTradeMarketRuleDsl._

withTrade(
  200 discount_bonds IBM
    for_client NOMURA
      on NYSE
        at 72.ccy(USD)) {trade =>
  Mailer(user) mail trade
  Logger log trade
} cashValue
```

本节用了装饰器（Decorator）模式（参见6.10节文献[3]）作为组合手段。我们将`semantics`作为待装饰的DSL，在它的外面包裹装点其他必要逻辑。通过装饰器模式，对象可以动态地增减其职责。它的这种能力在我们需要组合有亲缘关系的DSL时，正好能派上用场。

要是待组合的几种语言之间没有亲缘关系，又会出现什么情况呢？日期和时间、货币、几何图形等领域的DSL，常常作为辅助工具穿插于更大型DSL的舞台间隙。下一节学习如何平稳掌控这类语言的成长变化。

6.7.2 层级关系的组合方式

在大型DSL脚本里面嵌入小型DSL是相当常见的做法。就以金融交易系统来说，例如处理货币换算，管理日期时间，投资组合报表中管理客户收支结余等场合，都不难发现DSL的身影。

6

现在假设我们需要为客户投资组合报表实现一种DSL，用于报告到给定日期为止，客户账户下持有的各类证券和现金结余。注意，“客户投资组合”和“结余”这两个词代表着两个重要的领域概念，值得我们用DSL的方式建模它们的抽象。它们虽然是两个互相独立的抽象，却时常发生一些密切的联系。

1. 避免与实现发生耦合

表6-7能帮我们理清这两个抽象之间的关联，只有掌握它们的关系，才能保证概念上独立的DSL在实现上也能保持独立。

`Balance`是盖在实现外面的抽象接口。Scala允许定义“类型同义词”（type synonym）。我们只要规定`type Balance = BigDecimal`，就可以用`Balance`这个名字来称呼客户名下的资产净值。但是这种便利会产生别的影响吗？抽象的`Portfolio` DSL会根据需要被特化为各种具体类型，形成6.7.1节`TradeDsl`那样的大家族。在这种情况下，直接在`Portfolio` DSL基类型中嵌入`Balance`的实现，将导致整棵`Portfolio`家族树都与内嵌的`Balance`具体实现耦合在一起。就算以后真的有需要，也绝无可能更改内嵌的实现。所以，一定不要将一方直接内嵌到另一方，而

应该从层次化的角度去设计组合形态。最终让两个DSL既能完美契合，又不至于密不可分，随时能换上你想要的实现。

表6-7　按照层级关系组合多个DSL

需要各自独立发展的两个关联抽象	
“结余”指的是：	“客户资产组合”指的是：
■ 客户持有的现金和证券数量	■ 反映客户拥有的各类资产结余的报表
■ 一个具有确切语义的重要领域概念，可以被建模为DSL	■ 一个具有确切语义的重要领域概念，可以被建模为DSL
■ 一个数额，在实现中可以用BigDecimal来表示，但BigDecimal对于领域用户来说不具有任何意义	

注意：

你应该时刻注意，不要让对外公开的语言结构暴露了背后的实现。能做到这一点的话，不仅DSL的可读性更好，还便于在不影响客户代码的前提下，完美地更改内部实现。关于如何隐藏内部实现的话题，请参考附录A中的讨论。

对关系建模：

下面这段脚本清楚地说明，在领域用户眼中，结余抽象和资产组合抽象应该在领域API里面呈现什么样的关系：

```
trait Portfolio {
  def currentPortfolio(account: Account): Balance
}
```

待完成工作：

结余DSL可以有多种实现，资产组合DSL也一样，我们设计的组合方式，要保证两者的关联关系不因为任何一方的实现变化而受到影响。定义一个`Portfolio` DSL实现时，应该能够灵活地插入任意一种`Balance` DSL实现。

代码清单6-19的DSL用于对客户资产组合建模，想想看它有什么问题。

代码清单6-19　存在实现耦合的DSL

```
package dsl

import api._
import api.Util._

trait Portfolio {
  type Balance = BigDecimal                              ❶ 内嵌的实现
  def currentPortfolio(account: Account): Balance
}

trait ClientPortfolio extends Portfolio {
  override def currentPortfolio(account: Account) =      ❷ 现实中此处逻辑会很复杂
    BigDecimal(1200)
}
```

哈！才要实现第一个特化的`Portfolio` DSL，被内嵌绑住手脚的`Balance`抽象❶就支持不住了❷。我们试一试维持它们的层级关系不变，但是将`Balance` DSL的具体实现留在`Portfolio` DSL之外。虽然按照层级关系来组合，必然意味着一方要包含在另一方里面，但我们计划中的组合方式有一个地方不同于代码清单6-19，那就是我们嵌入的是DSL的接口，而非实现。答案一想便知，没错，正是抽象val！我们让`Portfolio` DSL包含`Balance` DSL的一个实例，不妨

称为Balances。

2. 对结余的建模

太简单的DSL示例很难让你深刻理解DSL。你有当你看到DSL将底层复杂性用容易理解的语法表述，才能切实感受到DSL的强大表现力。前面我们简单地用BigDecimal来建模“结余”概念。但是对于熟悉证券交易操作的业内人士来说，客户账户下的结余实际上表示，客户在特定日期按照指定货币计算的现金头寸。例子将省略从客户的资产组合算出结余的具体过程。作为DSL契约的Balances如代码清单6-20所示，同时列出的还有它的一个具体实现BalancesImpl。

代码清单6-20 建模账户结余的DSL

```
package dsl

import java.util.Date
import api._
import api.Util._
import api.Currency._

trait Balances {                                          抽象类型
  type Balance

  def balance(amount: BigDecimal, ccy: Currency, asOf: Date): Balance
  def inBaseCurrency(b: Balance): (Balance, Currency)
  def getBaseCurrency: Currency = USD
  def getConversionFactor(c: Currency) = 0.9
}
                                                          ❶ 具体实现
class BalancesImpl extends Balances {
  case class BalanceRep(amount: BigDecimal,
    ccy: Currency, asOfDate: Date)
  type Balance = BalanceRep

  override def balance(amount: BigDecimal, ccy: Currency, asOf: Date)
    = BalanceRep(amount, ccy, asOf)

  override def inBaseCurrency(b: Balance)
    = (BalanceRep(b.amount * getConversionFactor(getBaseCurrency),
         b.ccy, b.asOfDate), getBaseCurrency)
}

object Balances extends BalancesImpl
```

客户可以根据喜好指定报告结余金额时采用的货币类型，而监管机构往往要求一律按照基本货币来计算。基本货币是投资者记账时采用的货币。外汇市场上一般用美元充当基本货币。在代码清单6-20的DSL实现中，inBaseCurrency方法负责按基本货币报告结余。我们在Balances的示例实现BalancesImpl里面，将抽象类型Balance落实为由金额、货币、日期（结余总是针对具体日期进行计算）构成的三元组。

3. 结余DSL与资产组合DSL的组合

Portfolio DSL需要为组合做一些准备，安排一个数据成员的位置给Balances类型的抽象val❶，如代码清单6-21所示。

代码清单6-21　资产组合DSL的接口契约

```
package dsl

import api._

trait Portfolio {
  val bal: Balances                    ❶ 为结余DSL预留的抽象val
  import bal._                         ❷ 对象导入语法
 def currentPortfolio(account: Account): Balance
}
```

为了在类的内部访问bal对象的所有成员，我们运用了Scala的对象导入（object import）语法❷。定义完接口，我们再看看具体实现。代码清单6-22实现了一个计算客户账户结余的Portfolio。

代码清单6-22　资产组合DSL的一个具体实现

```
trait ClientPortfolio extends Portfolio {
  val bal = new BalancesImpl           落实到具体的实现
  import bal._

  override def currentPortfolio(account: Account) =
    val amount = //..                  实现细节略
    val ccy = //..
    val asOfDate = //..

    balance(amount, ccy, asOfDate)
}

object ClientPortfolio extends ClientPortfolio
```

例中的ClientPortfolio DSL已经落实到Balances的一个具体实现。下一步需要保证，当ClientPortfolio与其他同样用到Balances的DSL组合时，双方拥有相同的Balances实现。

我们用另一个例子来说明怎样做到这一点。代码清单6-23给出了一个起装饰器作用的Portfolio实现——Auditing，它可以为其他Portfolio实现增加审计功能。

代码清单6-23　资产组合DSL的另一个实现

```
trait Auditing extends Portfolio {
  val semantics: Portfolio             ❶ 内嵌的资产组合DSL

  val bal: semantics.bal.type          ❷ Scala单例类型
  import bal._

  override def currentPortfolio(account: Account) =
    inBaseCurrency(                    按基本货币报告结余
      semantics.currentPortfolio(account))._1
}
```

Auditing不但能与其他Portfolio DSL进行组合❶，还保证被组合的Portfolio（即semantics）与它嵌入到自身的Balances DSL❷使用同一个实现。（Balances嵌入到Auditing的父类Portfolio。）我们为了施加这样的约束，将Auditing的成员bal声明为semantics.bal，从而将它定义为一个Scala单例类型。接下来，可以指定semantics和bal指

定具体的实现，创建一个支持`Auditing`功能的`ClientPortfolio`抽象。请看下面的代码片段：

```
object ClientPortfolioAuditing extends Auditing {
  val semantics = ClientPortfolio
  val bal: semantics.bal.type = semantics.bal
}
```

通过层级方式来组合多个DSL，其优点如表6-8所示。

表6-8　通过层级方式来组合DSL的优点

优　　点	理　　由
不受抽象内部表达的影响	对多个DSL进行组合时，语句中不会出现内嵌实现的任何细节
耦合松散	参与组合的DSL之间耦合松散，各自可以独立演进
静态类型安全	Scala拥有强大的类型系统，可以由编译器来实施所有的约束

DSL组合这一主题在“Polymorphic embedding of DSLs”（6.10节文献[7]）这篇论文中有详细介绍。如果希望详细了解在Scala语言中组合DSL的其他方法，请参阅该论文。

发挥Scala语言面向对象编程和函数式编程的双重能力，灵活组合各种抽象的技巧，本章至此已经展示了很多示例，但我们对组合这个话题的讨论还缺一角，那就是Monad化抽象。Monad化抽象广泛用于构建具备组合性的计算结构。Monad概念源自范畴论（category theory）（别紧张，这本书不会涉及Monad背后的数学理论；感兴趣的话，你可以自行研究）。下一节将展示怎样利用Scala语言的Monad化结构去实现一些语法糖。这种技术可用于设计DSL操作的串联机制。

6.8　DSL 中的 Monad 化结构

我一直反复强调，抽象组合得越好，DSL的可读性就越强。当针对“运算”进行抽象时，函数式编程提供的组合能力要超过面向对象编程模型。这是因为函数式编程将各种运算都当做纯数学函数来使用，不产生改变状态的副作用。如果函数与改变状态分离，它就成了一种不依赖于任何外部上下文，可以单独验证的抽象。借助函数式编程提供的数学模型，运算可以被组合成一些函数式的组合体。我不准备深入介绍范畴论或者其他具有类似功用的数学理论体系。你只要记住，函数的组合性意味着我们可以用简单的构件块搭建出复杂的抽象。

1. 什么是Monad

Monad可以理解成*函数组合或者加强版的绑定*。按照Monad的规则构造出来的抽象，可以在优美的组合语义指挥之下，用来构造更高阶的抽象。



将运算的结构使用“值”和“使用这些值的运算序列”来表示，我们就得到一个Monad抽象。许多Monad再按照依存关系组合起来，可以构成更大规模的运算。Monad的理论基础是令人望而生畏的范畴论，但如果你有兴趣了解，那么6.10节文献[10]可以作为初步的阅读材料。一般读者并不需要深究，放松就好。

这里的讨论不会深入到理论层面，只会针对设计Scala DSL的需要，从实用的角度探讨Monad的一些特性。等到第8章介绍Scala的分析器组合子时，再展示更多的Monad化构造单元。本节讨论的主要对象是Scala语言中一些具备Monad性质的操作，它们可以使DSL的组织比面向对象编程中的对应结构更加优美。

本节的补充内容简单介绍了Monad。Monad概念的详细信息请参阅6.10节文献[9]。

Monad小讲座

Monad是一种可以绑定一系列运算的抽象。这里没有给出理论化的一般性定义，而是试着用Scala的语汇来界定Monad有哪些性质。（如果按照传统思路，需要动用范畴论或Haskell语言，从一阶逻辑的基本原理说起；以Scala语言作为解释基础似乎更实用一些）。

一个Monad由以下三部分定义。

(1) 一个抽象`M[A]`，其中`M`是类型构造函数。在Scala语言中可以写成`class M[A]`，或者`case class M[A]`，又或者`trait M[A]`。

(2) 一个`unit`方法（`unit v`）。对应Scala中的构造函数`new M(v)`或者`M(v)`的调用。

(3) 一个`bind`方法，起到将运算排成序列的作用。在Scala中通过`flatMap`组合子来实现。`bind f m`对应的Scala语句是`m flatMap f`。

例如`List[A]`是Scala语言中的一个Monad。它的`unit`方法由构造函数`List(…)`定义，它的`bind`方法则由`flatMap`组合子实现。

那么是不是任何抽象只要具备以上三部分，就成了一个Monad呢？不一定。Monad还必须满足以下三条规则。

(1) **右单位元**（identity）。即对于任意Monad `m`，有`m flatMap unit => m`。以Scala的`List` Monad来说，我们可以得出`List(1,2,3) flatMap {x => List(x)} == List(1,2,3)`。

(2) **左单位元**（unit）。即对于任意Monad `m`，有`unit(v) flatMap f => f(v)`。换成Scala的`List` Monad，这个关系意味着`List(100) flatMap {x => f(x)} == f(100)`，其中`f`返回一个`List`。

(3) **结合律**。即对于任意Monad `m`，有`m flatMap g flatMap h => m flatMap {x => g(x) flatMap h}`。这个定律告诉我们运算的结果取决于运算顺序，但不受嵌套的影响。作为练习，读者可以在Scala `List`身上验证一下这个定律。

2. Monad如何降低非本质复杂性

代码清单6-24中使用Java编写的例子是Web交易应用中的一个典型操作。我们凭一个键从`HttpRequest`或`HttpSession`中取出对应的值。我们取出来的这个值是某一笔交易的编号，用这个编号可以从数据库中查询到具体的交易，获得对应的`Trade`对象。

代码清单6-24 Java对运算中分支路径的处理方式

```
String param(String key) {
  //..
  return value;
}

Trade queryTrade(String ref) {
  //查询
  return trade
}

public static void main(String[] args) {
  String key;
  //设置要获取的键

  String refNo = param(key);
  if (refNo == null) {
    //异常处理
  }

  Trade trade = queryTrade (refNo);
  if (trade == null) {
    //异常处理
  }
}
```

从请求或会话获取参数值

执行数据库查询

❶ 检查空值

这段代码表现出一定程度的非本质复杂性❶，对抽象的表面语法造成了污染。在这段从上下文参数获取领域对象的运算中，每一步都要执行空值检查，且每次检查都是显式进行的。代码清单6-25中的Scala代码与上面的代码功能完全相同，但利用了Scala的Monad化语法结构`for` comprehension。

代码清单6-25 Scala的Monad化语法结构`for` comprehension

```
def param(key: String): Option[String] = {
  //..
}
def queryTrade(ref: String): Option[Trade] = {
  //..
}

def main(args: Array[String]) {
  val trade =
    (
      for {
        r <- param("refNo")
        t <- queryTrade (r)
      }
      yield t
  ) getOrElse error("not found")

//..
}
```

❶ Monad化的返回

❷ Monad化的返回值

`main`方法中的Monad化结构最终怎么串联起来，创建出`trade`对象，对此我们进行详细分析。

6

param返回的Option[String] ❶是一个Monad。按照Scala的设计，Option[]用于表示一则有可能不产生任何结果的运算。queryTrade返回的Option[Trade] ❶也是一个Monad，只不过它的类型不同于Option[String]。我们希望两步运算的串联满足一定的条件，即当param返回空值时，queryTrade必须不被调用。代码清单6-24用了显式的空值检查来实现这样的条件。而这里因为利用了Monad化结构，让Option[]在其实现内部负责空值检查的例行工作，表面上的代码得以保持简介，摆脱了非本质复杂性❷。

Monad是怎么串联两步运算的呢？是通过本节补充内容中介绍的Monad三要素之一——bind操作。bind操作在Scala语言中被实现为flatMap组合子，而for comprehension❷只不过是包裹在flatMap外面的一层语法糖，从下面的代码片段可以看穿这一点。

剥掉for comprehension糖衣，里面掩盖着的bind操作就清楚地显露出来了。

```
param("refNo") flatMap {r =>
  queryTrade(r) map {t =>
    t}} getOrElse error("not found")
```

flatMap组合子（等价于Haskell的>>=操作）在片段中起到一种承上启下的作用。重要的bind操作将param的输出对接到queryTrade的输入，同时在操作内部处理所有必要的空值检查。for comprehension在flatMap组合子的基础上提供更高阶的抽象，进一步改善DSL的可读性和表达能力。

对Monad、flatMap和for comprehension的深入讨论超出了本书的范围。目前来说，只要知道Monad化结构和操作能简化DSL的实现就可以了。我们刚刚用Monad作为手段，在不引入非本质复杂性的前提下，对存在依赖关系的运算进行排序。这只是Monad最普通不过的一种用法而已。除此之外，Monad还广泛用作一种机制，解释纯函数式语言的副作用，处理状态变化、异常、continuation等运算。显然，Monad的这些用法都可以用来改善DSL的表现力。Scala语言中Monad的更多详细信息请参考6.10节文献[10]。

为了给讨论画上一个漂亮的句号，我们最后用一个例子来说明Monad如何提高DSL抽象的表现力。这个例子是对代码清单6-20中交易处理流程DSL的重新实现，但我们用Monad化的for comprehension取代原来的偏函数来排列操作的顺序。这个练习目的是让你学会用Monad的思维去构建DSL。我们会尽量让例子保持简单，仅针对性地演示应该怎样设计各步运算，以便于通过for comprehension进行串联。

3. 设计Monad化的交易DSL

不再多说，我们这就换一种方式重新定义代码清单6-17的TradeDsl。修改后，所有的流程方法都不再返回PartialFunction，而是分别返回一个Monad（Option[]）。

代码清单6-26　Monad化的TradeDsl

```
package monad

import api._
class TradeDslM {
  def validate(trade: Trade): Option[Trade] = //..
  def enrich(trade: Trade): Option[Trade] = //..
```

```
  def journalize(trade: Trade): Option[Trade] = //..
}

object TradeDslM extends TradeDslM
```

用一个for comprehension套住上面的DSL，即可对一个交易的集合调用流程方法组成的序列：

```
import TradeDslM._

val trd =
  for {
    trade <- trades
    trValidated <- validate(trade)
    trEnriched <- enrich(trValidated)
    trFinal <- journalize(trEnriched)
  }
  yield trFinal
```

除了改用Monad的绑定操作来串联各步骤，这段代码的功能与代码清单6-20相同。原先基于偏函数的实现只能串联类型完全一致的操作。而这段for comprehension里面的操作序列，前后操作的类型并不完全一致。trades是Iterable类型的List，每次迭代它都产生一个Trade对象。我们并不需要特意检查列表的结尾,因为List也像Option[]一样是个Monad,其内部的flatMap组合子会处理好类似的边界条件。validate返回一个Option[Trade]，可能是Some(trade)，也可能是None。当validate的输出被送入enrich时，不需要做任何显式的空值检查，也不需要进行任何Option[Trade] -> Trade显式转换。只要串联用的管道是List[]、Option[]等Monad化构造，flatMap组合子会自动完成所有的绑定。从这个意义上说，通过Monad绑定来串联操作，比代码清单6-17通过偏函数来串联效果更好。如果设计得当，Monad化的操作可以成就表现力强的DSL，配合Scala的for表达式（或Haskell的do notation）语法糖一起使用，效果尤佳。

如果你想知道上面片段的flatMap展开形式，它是下面这个样子的：

```
trades flatMap {trade =>
  validate(trade) flatMap {trValidated =>
    enrich(trValidated) flatMap {trEnriched =>
      journalize(trEnriched) map {trFinal =>
        trFinal
      }
    }
  }
}
```

显然这种写法的编程味道要浓很多，还是之前用for语句的版本更适合领域用户阅读。

读完本节，你知道Monad是Scala提供的另一种抽象组织手段。它与偏函数有细微的差别。它能用一种纯数学的方式，把抽象按照依赖关系串联起来。Scala自带了不少Monad化的构造单元，设计DSL时别忘了善用这些素材。

6.9 小结

Scala社群热衷于DSL事出有因。Scala作为现今最具影响力的编程语言之一，为设计富有表现力的DSL提供了一流的支持。

本章已经逐一展示能用于内部DSL设计的Scala语言特性。我们从一份Scala特性名单开始，然后通过分析证券交易领域的众多DSL片段，认真深入分析这些DSL片段的设计。从结构上说，DSL是底层实现模型上的一重门面。本章的讨论焦点在领域模型与它上的语言抽象之间来回切换。

DSL需要在契约层次上表现出它的组合能力，避免暴露任何实现细节。你的设计从一开始就要遵循这样的原则，因为不同的DSL有不同的发展步调，需要修改某一个DSL的实现时，不要影响其他DSL或者核心应用。我们还介绍了不同的Scala内部DSL之间，怎样利用Scala强大的类型系统，静态地组合到一起。在最后阶段的讨论中，我们观察了Monad化的操作怎样帮助创建具有组合能力的DSL构造单元。Monad在Scala语言中的位置并不像它在Haskell编程模型中的地位那么显著，但用处并不小。Scala的Monad化语法`for` comprehension可以用来排列领域操作的顺序，同时不引入过多的非本质复杂性。

要点与最佳实践

- **Scala语法简练，可省略句末分号，具备类型推断特性**。这些特性可以使DSL的表面语法保持紧凑。
- **Scala有丰富的语言构造可作为设计抽象的手段**，请充分利用类、trait、对象等元件去提高DSL实现的扩展能力。
- **Scala提供了强大的函数式编程能力**。请善用这种能力来对DSL中的行为进行抽象。摆脱对象范式的锢囿，使DSL实现对用户的表现力更强。

内部DSL需要一种宿主语言，有时会受到宿主语言能力的限制。打破这些限制的方法之一是自己设计一种外部DSL。下一章将探讨外部DSL的若干基本构件。从编译器和语法分析器的一些基本理论入手，然后过渡到分析器组合子（parser combinator）等目前广泛使用的高阶结构。敬请期待！

6.10　参考文献

[1] Odersky, Martin, and Matthias Zenger. 2005. Scalable component abstractions. *Proceedings of the 20th annual ACM SIGPLAN conference on object-oriented programming systems, languages, and applications*, pp 41-57.

[2] Wampler, Dean, and Alex Payne. 2009. *Programming Scala: Scalability = Functional Programming + Objects*. O'Reilly Media.

[3] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional.

[4] Ford, Neal, *Advanced DSLs in Ruby*, http://github.com/nealford/presentations/tree/master.

[5] Emir, Burak, Martin Odersky, and John Williams. Matching Objects With Patterns. *LAMP-REPORT-2006-006*. http://lamp.epfl.ch/~emir/written/Matching-ObjectsWithPatterns-TR.pdf.

[6] Evans, Eric. 2003. *Domain-Driven Design: Tackling complexity in the heart of software*. Addison-Wesley Professional.

[7] Hofer, Christian, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic Embedding of DSLs. *Proceedings of the 7th international conference on generative programming and component engineering*, 2008, pp 137-148.

[8] ScalaTest. http://www.scalatest.org.

[9] Wadler, Philip. 1992. The essence of functional programming. *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on principles of programming languages*. pp 1-14.

[10] Emir, Burak. Monads in Scala. http://lamp.epfl.ch/~emir/bqbase/2005/01/20/monad.html.

[11] Pierce, Benjamin C. 1991. *Basic Category Theory for Computer Scientists*. The MIT Press.

[12] Ghosh, Debasish. Implementation Inheritance with Mixins—Some Thoughts. *Ruminations of a Programmer*. http://debasishg.blogspot.com/2008/02/implementation-inheritance-with-mixins.html.

[13] Venners, Bill. Abstract Type Members versus Generic Type Parameters in Scala. http://www.artima.com/weblogs/viewpost.jsp?thread=270195.

[14] Ghosh, Debasish. Scala Self-Type Annotations for Constrained Orthogonality. *Ruminations of a Programmer*. http://debasishg.blogspot.com/2010/02/scala-self-type-annotations-for.html.

第 7 章

外部DSL的实现载体

本章内容

- ❑ 外部DSL的处理流程
- ❑ 语法分析器的分类
- ❑ 用ANTLR开发一种外部DSL
- ❑ Eclipse Modeling Framework与Xtext

外部DSL跟内部DSL一样，都是在已有的领域模型外面覆盖一层抽象，差别在于怎样实现这层抽象。外部DSL会自行建立一套语言处理设施，包括语法分析器、词法分析器和处理逻辑。

我们的讨论将从外部DSL处理设施的整体架构开始。内部DSL可以借用宿主语言的基础设施，而外部DSL需要从头开始构建它们。第一步我们打算手工编写一个语法分析器，然后使用ANTLR语法分析器生成器开发你的第一个自定义外部DSL。本章的最后一节会介绍另一种外部DSL开发范式——在外部DSL开发环境Xtext的支持下，基于EMF（Eclipse Modeling Framework）的模型驱动方式。图7-1是本章讨论进程的路线图。

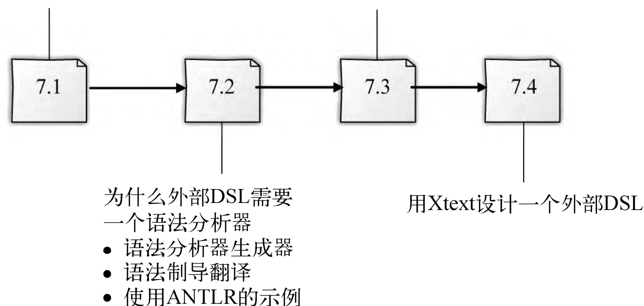


图7-1　本章路线图

本章将学习利用能从市面上获得的工具开发一套语言处理基础设施。有了这些基础设施，就可以用它开发自定义DSL语言处理程序。

7.1　解剖外部 DSL

1.5节粗略描绘过在自定义语言基础设施的基础上，构建外部DSL的情况。本节将从细节上探讨DSL的架构怎样随着它所描述的领域模型而发展变化，期间还会讨论供设计者选择的实现选项以及如何取舍。

7.1.1　最简单的实现形式

我们从外部DSL最简单的实现形式说起。DSL的自定义语法需要有配套的语法分析器。分析引擎首先对输入流进行词法分析，将其转化为可识别的词法单元（token）。词法单元在语法上也称为终结符号（terminal）。随后这些词法单元作为语法正确的语句，被送入产生式规则（production rule）进行处理。整个过程如图7-2所示。

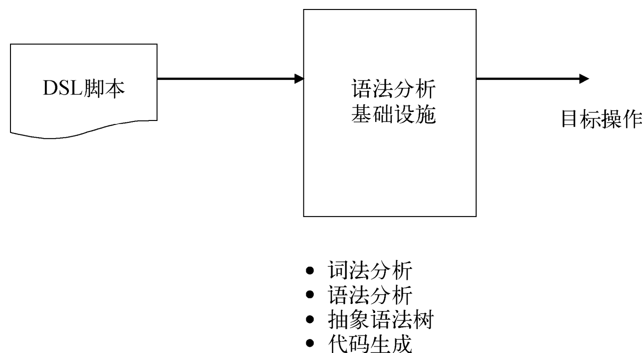


图7-2　外部DSL最简单的实现形式。语法分析基础设施包揽了产生目标操作所需的一切事务。DSL脚本的所有处理步骤（词法分析、语法分析、生成AST、生成代码）全部集中在一个构造块中

处理DSL脚本输入和生成必要输出所需的全部工作都由语法分析基础设施来执行。



DSL不一定需要非常复杂精密的语法分析基础设施。对于简单的领域语言，用字符串处理器、正则表达式处理器等简单的数据结构来充当语法分析引擎的做法并不罕见。此时将词法分析、语法分析、代码生成等操作步骤整合到一起往往是合理的。

图7-2的设计无法很好地适应较复杂的情况。在需求很简单，且复杂性不会增加的情况下，我们可以为语言处理基础设施选择这种大包大揽的实现形式。可惜世界上的问题不都是简单的。下一节将会介绍，解决复杂性的唯一途径是用不同模块实现不同任务，并且引入适当程度的抽象。

7.1.2　对领域模型进行抽象

图7-2中全能的语法分析基础设施把产生目标输出所需的一切处理工作都放在一个盒子里完

7

成。前面提到，这种设计难以适应语言复杂性增加的情况。以一个不算复杂的DSL来说，该设施至少需要在它的单个抽象单元内执行以下任务。

- 依据一套语法规则对输入进行分析。
- 语言经过分析后被保存为AST的形式。较为简单的场合可以省略生成AST的步骤，直接在语法中嵌入目标操作。
- 对AST进行标注，将其充实为中间表示，为执行目标操作做好准备。
- 处理AST，执行代码生成等目标操作。

让一个抽象单元承担这么多职责，负担实在太重了。我们想一想有没有解决的办法。

1. 模块化

我们可以试着从大盒子里分离出来一部分职责，让设计变得模块化一些。图7-3是一种分法。

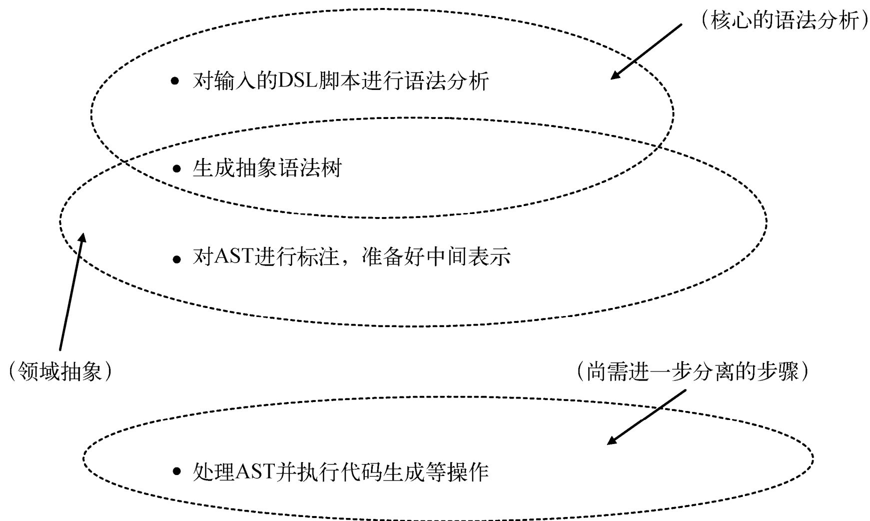


图7-3　对图7-2大中盒子包含的四项职责进行分解，分离任务。虚线围起来的部分分别代表一项功能

语法分析是图7-3的核心功能之一，应该用一个独立的抽象来表示。语法分析的结果之一自然是产生一棵AST。AST以一种独立于语言语法的形式，呈现语言的结构化形态。根据AST在下一阶段的用途和处理要求，我们需要为它增加其他信息，如对象类型、标注等上下文标记。增加了这些信息的AST逐渐积累语言的语义信息。

2. 语义模型

在为某一领域设计DSL的过程中，充实后的AST成为该领域的语义模型。图7-3前两部分之所以出现重叠，是因为核心语法分析过程要产出某种数据结构。

然后由下一阶段的处理过程向该数据结构注入领域知识。于是完整流程就如图7-4所示，我们可以将领域的语义模型作为DSL处理流程中的一个核心抽象。

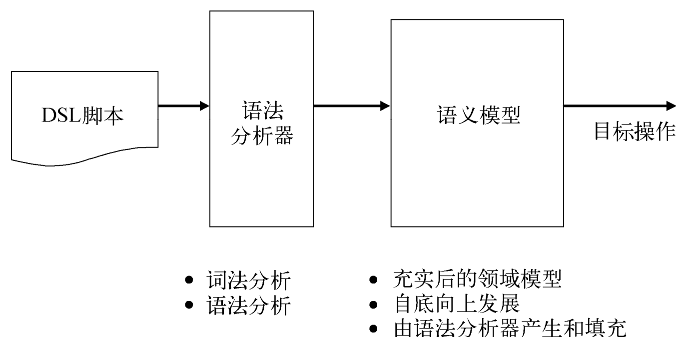


图7-4 我们将图7-2的语法分析设施基础设施盒子拆分成两个抽象。语法分析器负责核心的语法分析。语义模型从语法分析引擎中独立出来，成为单独的抽象。语义模型封装了所有的领域相关事项，预备提交给后续负责生成目标操作的设施

语义模型是DSL脚本处理后产生的、增加了领域语义的数据结构。它的结构与DSL的语法无关，更多地反映了系统的解答域模型。语义模型作为一层完美的抽象，分离了输入的语法导向的脚本结构与另一边的目标操作。

DSL处理流程的目标输出有很多功能。它可以直接生成应用代码。它也可以生成一些资源，供应用运行时使用和解释，比如Hibernate用来产生数据模型的对象-关系映射文件。Hibernate是一种ORM（对象–关系–映射）框架。详情请参阅http://www.hibernate.org。语义模型使上下层保持分离，同时独自充当所有必要领域功能的供应仓库。

拥有一个设计得当的语义模型，对于提高应用的可测试性大有好处。因为我们可以脱离DSL的语法层，单独测试应用的整个领域模型。下面就来更详细地讨论一下语义模型，观察它是怎样在外部DSL的开发周期中逐渐形成的。

3. 填充语义模型

语义模型是供应领域模型的仓库。语法分析器一边消耗DSL脚本的输入流，一边填充语义模型。语义模型的设计完全独立于DSL语法，而且模型的构成方式和内部DSL一样，由一些更小的抽象自底向上组合起来。图7-5形象说明了语义模型怎样由下至上逐渐形成一个汇集领域结构、属性和行为的仓库。

外部DSL这种边做语法分析，边填充语义模型的方式，正是它与内部DSL的区别所在。我们在构造内部DSL时，先在宿主语言中建立较小的抽象，然后通过宿主语言本身的组合功能，建立更大的抽象。而对于外部DSL来说，对语言的语法分析与产生较小的抽象同步进行，分析树成长壮大，意味着语义模型凝聚了更多的血肉，成为领域知识的具体表示。

产生语义模型之后，我们用它来生成代码，操作数据库，或者继续生成其他应用组件所需的模型。现在回头看看图7-4的DSL处理架构，听完前面的讲解，你是否相信了拆分抽象的好处呢?

以架构的角度来说，内部和外部DSL都是建立在语义模型上面的一层抽象。内部DSL借用了宿主语言的语法分析器，公布给用户的契约只是包在语义模型外表的薄薄一层装饰。外部DSL需要自行构建相关设施去解析DSL脚本并执行一些操作，执行的结果是填充了语义模型。

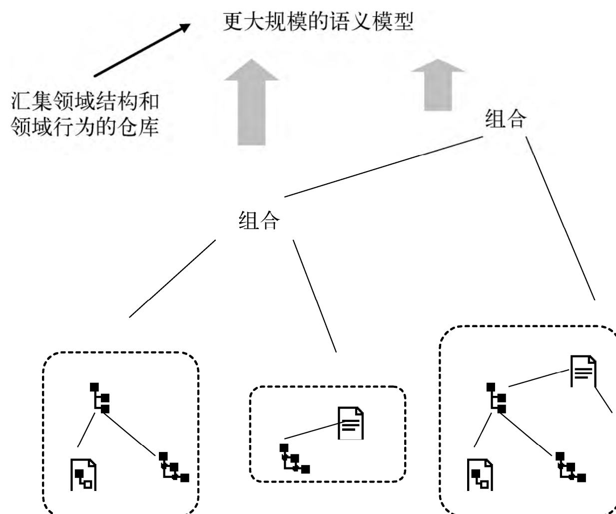


图7-5　语义模型由众多较小的领域抽象自底向上组合而成。我们先发展出各种领域实体的抽象，即虚线框中的小抽象单元，然后把它们一层一层地组合成更大的实体，最后得到一套完整的领域抽象，也就是我们的语义模型

语法分析器在外部DSL的处理设施中负责识别DSL脚本语法。各种形式的语法分析器和词法分析器需要用到不同的实现技术，掌握这些技术是学习外部DSL开发的重要一环。

下一节将讨论语法分析技术。我们并不打算详细论述语法分析器实现，而会在大致了解之后，介绍几种语法分析技术及其所针对的语法类别。选择最合适的工具（如语法分析器生成器）开发外部DSL时，不见得需要完全了解分析器是怎么实现的。当然，设计不同类别的语言有不同程度的知识要求，多知道一些分析器的实现技术总是有用的。本章末尾列出的参考文献可以作为学习这方面知识的向导。

7.2　语法分析器在外部 DSL 设计中的作用

待执行的DSL脚本被送入词法分析器，经过词法分析器的处理，输入流被划分为语法分析器能理解的的可识别单元。当语法分析器顺利处理完全部输入流，到达一个成功的终结状态，我们就说该语法分析器识别了输入的语言。图7-6是这个过程的图解。

一提起词法分析器和语法分析器，我们总是不由自主地想得很复杂，实际上并非如此。它们的复杂度取决于你所设计的语言。前面提到，假如DSL足够简单，我们甚至不必区分词法分析阶段和语法分析阶段。一个通过正则表达式来操作调整输入脚本的字符串处理器，就足以承担全部的解析工作。简单的语言可以靠手工编写分析器，与之相对，更复杂一些的语言需要借助一些专业的开发设施。下面就介绍如何使用生成语法解析器的基础设施，来构建面向复杂DSL的语法解析器。

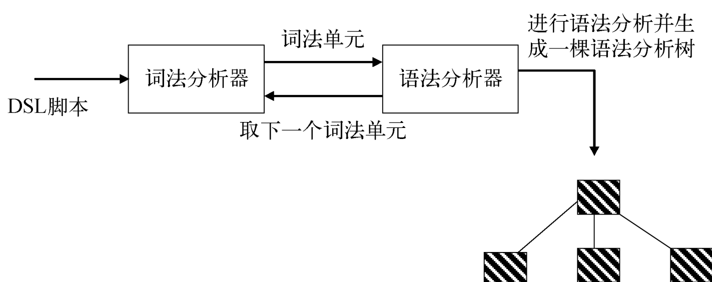


图7-6 语法分析过程。DSL脚本被送入词法分析器去划分词法单元，结果送入语法分析器

7.2.1 语法分析器、语法分析器生成器

我们所设计的语法分析器，实质是对语言语法的一种抽象。如果我们打算手工编写整个分析器，那么需要做这两件事情：

- 定义语言的BNF语法；
- 编写与该语法对应的语法分析器。

然而手工编写有一个弊病，这样写出来的全部语法都嵌入到了代码中。对语法的任何修改，都意味着也要对相应的实现代码进行大幅修改。这种情况是实施编程的抽象层次过低的典型表现（附录A有详细解释）。

对于较复杂的语法分析器，利用语法分析器生成器要比直接手写更好一些。语法分析器生成器可以提高我们实施编程的抽象层次。我们只需要定义这两样东西：

- 按Extended Backus Naur Form（EBNF）语法格式书写的语法规则；
- 当语法规则识别成功时，希望执行的自定义操作。

在运用生成器的情况下，实现自定义语法分析器的基础代码完全封装在生成器内部。错误处理、生成分析树等一般事务成为内建在生成器内部的标准例程，无论我们创建什么样子的语法分析器，这些部分都相同。

语法分析器生成器作为一种提高抽象层次的技术，其优点首先是减少代码量，减轻编写、管理、维护的负担。另外，很多生成器能够生成多种目标语言下的语法分析器，这也是重要的优点。表7-1汇总了目前常用的几种生成器。

除了表7-1列出来的这几种生成器，还有原Sun Microsystems公司开发的Java Compiler Compiler（JavaCC，见https://javacc.dev.java.net/）和IBM公司开发的Jikes Parser Generator（见http://www10.software.ibm.com/developerworks/opensource/jikes/project/）。Jike和JavaCC生成的都是Java代码的语法分析器，其功能也都类似于YACC和Bison。

无论产生语法分析器的途径是手工编写，还是由生成器产生，指导语法分析器行为的始终是你所用语言的语法。当分析器成功识别了语言，它会产生一棵语法分析树，将整个识别过程封装到这个递归的数据结构里面。如果我们在语法规则上附加了自定义动作，那么最后生成的分析树也会增加这些额外信息，形成语义模型。接下来我们用ANTLR做一次示范，看看生成器怎样把自定义语法翻译成领域语义模型。

表7-1　当前存在的语法分析器生成器

语法分析器生成器	相应的词法分析器	详　　情
YACC	LEX	属于UNIX发布版的一部分（最早开发于1975年），生成C语言编写的语法分析器
Bison	Flex	属于GNU发布版的一部分，功能几乎与YACC和LEX相同，但能生成C++语言编写的语法分析器
ANTLR（见http://antlr.org）	已包含在ANTLR内	由Terrance Parr开发。能生成多种语言编写的语法解析器，包括Java、C、C++、Python、Ruby等语言
Coco/R	自动生成词法扫描器（scanner）	Coco/R是一种编译器生成器，将一种源语言的属性语法（attributed grammar）输入给它，它会生成该语言的词法扫描器（scanner）和语法分析器

7.2.2　语法制导翻译

外部DSL实现在处理一段脚本时，首先要识别脚本中的语法，然后经过解析和翻译，产生语义模型。语义模型充当领域操作的仓库，供给后续的程序使用。那么如何识别语法呢？表7-2说明，要想成功识别，我们需要准备两套材料。

表7-2　识别DSL语法

材　　料	作　　用
一套上下文无关语法。它的意义是决定哪些产生式是有效的	语法规定了撰写DSL的结构形式。只有按照文法规则定义写成的DSL脚本才是有效的 **注意：**本节的所有例子都用ANTLR生成器来定义语法
一套语义规则，作用对象是语法识别的符号的属性。这些规则在生成语义模型时发挥作用	在每一条语法规则里，我们可以进一步定义一些操作，当语法规则被识别时执行。操作可以是生成语法分析树，也可以是生成其他任意的触发行为，只要与被识别的规则相关即可。这些操作很容易定义。任何一种语法分析器生成器都允许在DSL语法的定义中嵌入其他语言的代码。例如ANTLR可以嵌入Java代码，YACC可以内嵌C代码

图7-7展示了语法规则及其附带的自定义操作如何经过语法分析器生成器的处理，最后变成语义模型。

那么，我们就拿不起眼的“交易指令处理DSL”做一次实习，用ANTLR生成器来实现其语言解析。练习中将定义词法分析器和语法，并且在语法定义里加入若干自定义操作，生成交易指令处理DSL的语义模型。

1. 预备ANTLR示例

我们要定义的语言类似于第2章用Groovy开发的交易指令处理DSL，而且还要更简单一些。这个例子的目的是演示通过语法分析器生成器开发外部DSL的步骤。我们的工作除了制定DSL的语法，还要建立语义模型，然后由语义模型来生成一个代表全部交易指令的自定义抽象。

假设用户向交易机构发出了一串交易指令，看上去是这个样子：

```
buy IBM @ 100 for NOMURA
sell GOOGLE @ limitprice = 70 for CHASE
```

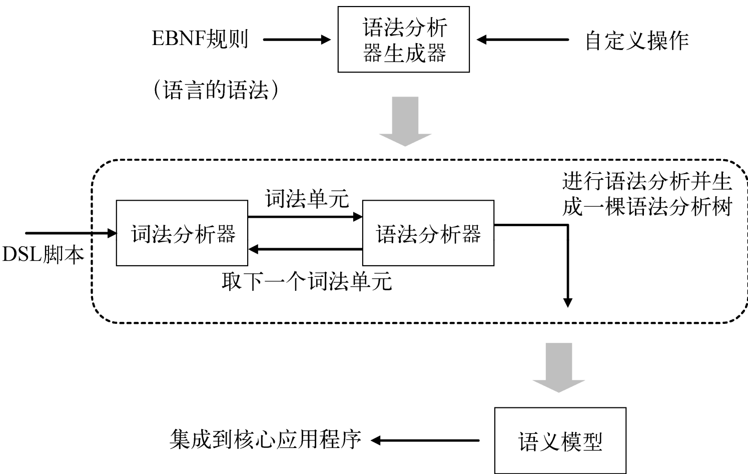


图7-7 语法分析器生成器的输入是语法规则及其附带的自定义。生成器随后生成词法分析器和语法分析器。DSL脚本经过词法和语法分析器的处理，形成语义模型。语义模型成为核心应用的一部分

整组指令由格式相同的若干行构成。我们首先要用ANTLR设计一个能处理这段脚本的外部DSL，然后生成适当的数据结构作为语义模型。简单起见，我们规定每一行代表一条交易指令，用户下达的全部指令构成一个指令列表集合。第一步从设计词法分析器开始。词法分析器的作用是对输入脚本进行预处理，使它成为一组可被语法识别的符号。

2. 设计词法分析器

词法分析器读入输入流，比照预设的词法单元定义，将输入流中的字符组合转换成一个个词法单元。利用ANTLR，可以在文法定义中直接内联词法规则，也可以将词法规则单独写到另外的文件。我们的例子单独用一个文件保存全部词法单元定义，文件名为OrderLexer.g。ANTLR用.g扩展名来表示文法定义文件（g是grammar的首字母）。请注意，代码清单7-1在书写词法单元定义时也用了一种DSL，其可读性和表现力都很优秀。

代码清单7-1 OrderLexer.g：用ANTLR编写的DSL词法分析器

```
lexer grammar OrderLexer;

EQ   : '=';
BUY  : 'buy';
SELL : 'sell';
AT   : '@';
FOR  : 'for';
LPRICE  : 'limitprice';
ID      : ('a'..'z'|'A'..'Z')+;
INT     : '0'..'9'+;
NEWLINE : '\r'? '\n';
WS      : (' '|'\t')+ {skip();};          <┘ 跳过空格
```

词法规则按“贪婪”方式匹配。分析器在对输入流进行匹配时，会选取匹配程度最高的规则。

假如遇到分不出高低的情况，那么分析器会按照规则在定义文件中的出现次序，选取最先出现的规则。

接下来我们转到另一个尚待实现的方面——语言的语法。语法由我们设计的语法规则来决定。

3. 设计语法规则

你希望DSL有什么样的语法，就定义什么样的语法规则。由于我们的DSL特别简单，而且只是用来演示，所以尽量省略错误处理方面的功能，着重突出语法规则的架构方面。读者可以从7.6节文献[2]了解ANTLR在语法规则定义上的详细做法，以及它为用户提供的各种灵活选项。

代码清单7-2定义的语法规则放在一个单独的文件OrderParser.g里面。文件中描述语法所用的标记方法，对于语言设计者来说，是表达能力极为出色的EBNF语法标记。当将词法分析器和语法分析器与驱动语法分析器的处理程序代码整合在一起时，你就会发现ANTLR如何通过这些EBNF描述，生成真正的语法分析器代码。生成语法分析器涉及的所有繁重事务都由ANTLR生成器代劳。开发者只需要专心定义好语言的语法即可。

代码清单7-2　OrderParser.g：用ANTLR编写的DSL文法规则

```
parser grammar OrderParser;

options {
  tokenVocab = OrderLexer;          ❶ 词法分析器的引用
}
                                    ❷ 全部交易指令
orders      : order+ EOF;
order       : line NEWLINE;
line        : (BUY | SELL) security price account;   ❸ 每条交易指令独占一行
security    : ID;
limitprice  : LPRICE EQ INT;
price       : AT (INT | limitprice);
account     : FOR ID;
```

熟悉EBNF语法记号的读者会觉得这些文法规则很好理解。我们希望建立一个交易指令的集合❷。每则指令占一行，说明下达的指令详情❸。语法定义文件的开头部分有一个指向词法分析器类的引用，放在`options`块内❶。

ANTLR带有一个图形界面的语言解释环境（ANTLRWorks，见http://www.antlr.org/works），允许用户通过规定的语法交互地运行示例DSL脚本。该环境会帮我们建立分析树。如果文法规则的解析出现异常，我们还可以在该环境内进行调试。

代码清单7-2指定的语法并没有包含任何语法制导翻译方面的自定义操作。这是故意为之，目的是让你领略一下ANTLR语法定义提供的DSL语法多么轻巧灵便。只要准备好语法规则，就可以成功识别一段有效的DSL脚本，并不需要其他任何东西！下一节将学习怎样在语法规则中嵌入Java代码来执行自定义操作。

4. 嵌入其他语言的代码作为自定义操作

像代码清单7-2那样的文法规则，ANTLR通过它生成的分析器，可以构建一棵默认的分析树。如果希望在分析树上增加其他信息，或者希望经通过分析DSL脚本生成另一个语义模型，那么可以采用内嵌其他语言代码的方式，于模型内添加自定义操作。下面就在代码清单7-2定

义的语法规则基础上，添加自定义操作代码，让DSL脚本在解析后生成一个自定义的Java对象集合。

内嵌代码首先需要定义`Order`对象，这是一个简单的Java对象（POJO）。解析脚本后，它会生成由一个`Order`对象列表构成的语义模型。代码清单7-3展示了嵌入语法规则中的最终操作。

代码清单7-3　OrderParser.g：语法规则中内嵌了执行自定义操作的代码

```
parser grammar OrderParser;

options {
  tokenVocab = OrderLexer;
}

@header {
  import java.util.List;
  import java.util.ArrayList;
}

@members {
  private List<Order> orders = new ArrayList<Order>();
  public List<Order> getOrders() {
    return orders;
  }
}

orders : order+ EOF;
order  : line NEWLINE {orders.add($line.value);};

line returns [Order value]
  : (e=BUY | e=SELL) security price account
    {
      $value = new Order($e.text, $security.value,
                 $price.value, $account.value);
    };

security returns [String value]: ID {$value = $ID.text;};

limitprice returns [int value]
  : LPRICE EQ INT {$value = Integer.parseInt($INT.text);};

price returns [int value] : AT
  (
    INT {$value = Integer.parseInt($INT.text);}
    |
    limitprice {$value = $limitprice.value;}
  );

account returns [String value] : FOR ID {$value = $ID.text;};
```

设定内嵌代码需要的导入声明和包声明

分析器类共享的代码

❶ 构造`Order`对象的集合

❷ 规则有返回值

如果读者不熟悉EBNF风格的语法规则描述方式，或者不清楚在规则中嵌入操作代码的写法，可以参阅7.6节文献[2]。注意，我们可以对规则定义返回值❷，返回值会跟着语法分析的进展向上传播。嵌套的运算一层层向上递进，最后形成一个`Order`对象的集合❶。

`Order`类的抽象如代码清单7-4所示，其代码出于演示目的已经尽量简化。

代码清单7-4　Order.java：Order抽象

```
public class Order {
  private String buySell;
  private String security;
  private int price;
  private String account;

  public Order(String bs, String sec, int p, String acc) {
    buySell = bs;
    security = sec;
    price = p;
    account = acc;
  }

  public String toString() {
    return new StringBuilder()
      .append("Order is ")
      .append(buySell)
      .append("/")
      .append(security)
      .append("/")
      .append(price)
      .append("/")
      .append(account)
      .toString();
  }
}
```

下一节要编写语言处理程序的主模块，由ANTLR生成的词法分析器、语法分析器，以及其他任何自定义的Java代码，都要在这里整合到一起。下面就来看看DSL脚本是怎样被解析并产生输出的。

5. 构建语法分析器模块

我们现在就可以用ANTLR构建分析器，与驱动代码进行集成。不过在此之前，还要先准备好驱动代码。驱动代码的任务是从输入中取得字符流，传递给词法分析器从而生成词法单元，再传递给语法分析器对DSL脚本进行识别。代码清单7-5中的Processor类就是这样的一段驱动代码。

代码清单7-5　Processor.java：分析器模块的驱动代码

```
import java.io.*;
import java.util.List;
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
public class Processor {

  public static void main(String[] args)                      驱动入口
    throws IOException, RecognitionException {
    List<Order> os =
      new Processor().processFile(args[0]);
    for(Order o : os) {                                       输出指令列表
      System.out.println(o);
    }
  }
```

```
  private List<Order> processFile(String filePath)
    throws IOException, RecognitionException {
    OrderParser p =
      new OrderParser(
        getTokenStream(new FileReader(filePath)));
    p.orders();
    return p.getOrders();
  }

  private CommonTokenStream getTokenStream(Reader reader)
    throws IOException {
    OrderLexer lexer =
      new OrderLexer(new ANTLRReaderStream(reader));
    return new CommonTokenStream(lexer);
  }
}
```

从文件中读入DSL脚本

❶ 语法分析器读入词法单元流

词法分析器生成的词法单元流

划分词法单元和读取输入流都利用了ANTLR的内建类。这段代码在构造语法分析器❶之后，随即在起始符号上调用了`orders()`方法，这个方法是ANTLR生成的。代码中引用的所有类（如`OrderLexer`、`CommonTokenStream`和`OrderParser`），要么是ANTLR根据语法规则生成的，要么来自ANTLR运行时。这段代码假设待执行的DSL脚本存放在一个文件里面，其路径作为命令行调用的第一个参数提供。

代码清单7-5还稍微演示了一下语义模型的用途，它输出语法分析过程中生成的`Order`对象列表。如果在真实的应用当中，我们可以将这些对象提供给系统中的其他模块，这样一来，DSL就与应用的核心集成在一起了。

这一节做了不少工作，现在不妨简要回顾一下。

6. 目前的成果

ANTLR提供了`org.antlr.Tool`等工具类，负责根据语法定义文件生成Java代码。然后所有的Java类，包括作为自定义代码一部分的Java类，都按照一般的构建过程那样编译就可以了。至此，处理外部DSL的基础设施就搭建完毕了，表7-3对我们已完成的工作做了一个小结。

表7-3 使用ANTLR语法分析器生成器构建外部DSL的步骤

步　骤	说　明
(1) 确定词法要素，为ANTLR准备词法分析器	代码清单7-1建立了词法分析器OrderLexer.g。词法单元定义沿用了之前的交易指令处理DSL
	注意：词法分析器的定义文件应该与语法分析器的定义分开。单独存储的词法定义可以跨语法分析器重用
(2) 用EBNF标记编写文法规则	代码清单7-2定义了DSL的语法，定义文件OrderParser.g按照ANTLR的语法写成
	文法规则的作用是识别有效的DSL语法，并在出现无效语法时给出异常
(3) 生成语义模型	代码清单7-3充实了语法规则的定义，通过插入自定义的Java代码，向语法分析过程中注入了语义操作。插入的一个个代码片段，实际上是搭建语义模型的部件
(4) 收尾	代码清单7-4的自定义Java代码完成对`Order`抽象的建模。代码清单7-5的驱动代码利用ANTLR的基础设施，将我们的DSL脚本送入前面建好的语法分析过程

经历了上面这些工作，我们应该对通过语法分析器生成器来构建和处理外部DSL的完整开发过程有了基本了解。利用生成器来实现外部DSL是非常常见的做法。如果你希望构建基于自定义基础设施的语言来设计外部DSL，那么使用生成器是最适合的。

ANTLR是一款十分优秀的的语法分析器生成器，广泛用于构建各式分析器和DSL。我们现在的成果只发挥了ANTLR的一小部分能力。ANTLR能处理许多种类的语法，而我们甚至连文法的分类都没介绍。理论上，如果对生成的分析器没有效率上的要求，我们其实可以分析任意语言。但实际上我们需要作出让步。开发DSL时，我们希望分析器能以最高效率处理我们的语言。一款能处理许多种语言的通用分析器虽然不坏，但假如它处理起我们的语言效率低下，那对于我们就没什么用处。ANTLR只是分析器生成工具中的一种，它所生成的分析器只能识别一部分特定类型的语言。

DSL的设计者需要清楚语法分析器的一般分类，每一种分析器的实现难点，还有每一种分析器能处理的语言类型。所以下一节将全面介绍语法分析器的分类及其实现复杂度。

7.3　语法分析器的分类

当语法分析器成功识别传递给它的输入流，就会为DSL脚本产生一棵完整的分析树。分析器依据分析树节点的构造顺序，分为不同类别。有的分析器从根节点开始构造分析树，称为*自顶向下*（top-down）分析器。有的分析器则相反，一开始先构造叶子节点，然后逐层构造根节点。这类分析器称为*自底向上*（bottom-up）分析器。自顶向下和自底向上这两类分析器的实现复杂度不同，它们所能识别的语法结构类型也不相同。DSL设计者至少应该大致了解每一类分析器的相关概念。图7-8分别展示了这两类分析器构造分析树的过程。

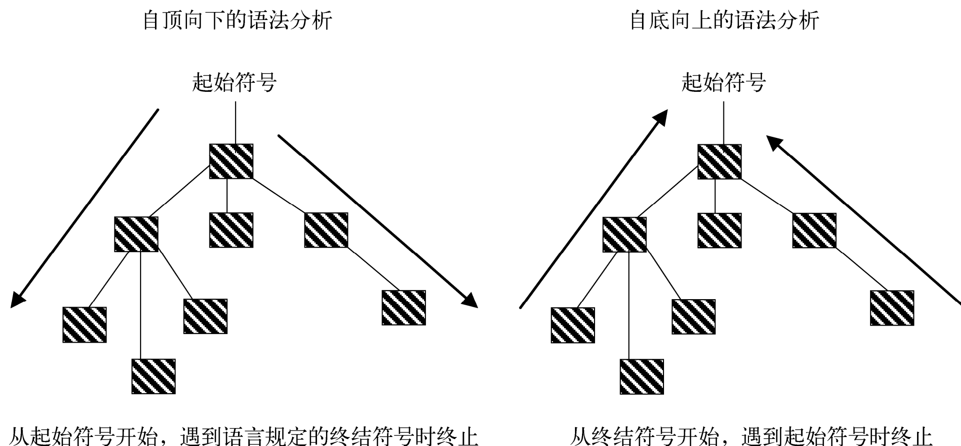


图7-8　自顶向下分析器和自底向上分析器构造分析树的过程

本节主要围绕分析树的构造方式，以及从产生式规则推导语言的方式，来讨论语法分析器的分类。在自顶向下和自底向上两大类别之下，存在着多种多样的变体，分别适合识别不同类型的语言结构，同时实现复杂度也有着相应的变化。这里假设读者已经基本了解语言处理方面的概念，

掌握语法分析技术、前瞻处理（look-ahead processing）、分析树等基础知识。如果需要了解这方面的背景知识，请参阅7.6节文献[3]。

首先介绍自顶向下分析器，学习其中一些常见的实现变体。

7.3.1 简单的自顶向下语法分析器

自顶向下分析器从*根节点*开始构造分析树，向输入流的*最左推导*（leftmost derivation）进行。也就是说，分析器处理输入的顺序是从左边的符号到右边的符号。

最常见的自顶向下分析器是RD（Recursive Descent，递归下降）分析器。递归下降这个名字应该如何理解呢？递归，说明分析器是通过一系列递归函数调用实现的（参见7.6节文献[1]）。下降指的是分析树的构造从树的顶部开始，与“自顶向下”的含义相同。

我们的学习过程将从最简单的RD分析器开始，然后逐步深入到其他功能更强大也更复杂的分析器，通过高效率的实现识别出更多的语言类。首先讨论LL(1)和LL(*k*) RD分析器。它们是自顶向下分析器中最简单的两种实现，涵盖了设计外部DSL所需的大部分功能。

1. LL(1)递归下降分析器

LL(1) RD分析器依靠一个前瞻词法单元完成对语法结构的分析。LL(1)这个名字应该怎样理解？第一个L表示分析器从左到右扫描输入字符串。第二个L表示当分析器自根至叶构造分析树时，产生子节点的次序是从左到右。最后括号内的1从分析器的定义就能猜到，它代表每步向前看一个符号。由于仅有唯一的前瞻符号，分析器将根据这个符号选择与其匹配的下一条产生式规则。

要是分析器找不到一条正好与前瞻符号完全匹配的产生式规则，会怎么样呢？这种情况有可能出现，语法里可能有多条产生式规则开头都是同一个符号。偏偏LL(1)分析器向前看的符号个数*仅为一个*。要想解决这种同一前瞻符号匹配多条规则的语法二义性问题，我们可以改用k>1的LL(*k*)分析器，也可以通过对语法定义提取左因子的方式，使之满足LL(1) RD分析器的要求（详见7.6节文献[1]）。

自顶向下分析器有时需要处理左递归的情况。例如语法中出现形如“A : A a | b”的规则，就会令自顶向下分析器陷入无限循环。前面提过，RD分析器通过一系列递归调用来实现。产生式规则中的左递归将使分析器永远递归下去。语法规则中的左递归可以通过规则变换来消除。7.6节文献[3]详细介绍了这方面的技术。

2. LL(*k*)递归下降分析器

LL(*k*)分析器与LL(1)分析器相似，但允许有更多的前瞻符号，因此功能更强大。它同样依靠其前瞻集来判断适用于输入符号的产生式规则。虽然更大的前瞻集意味着更复杂的分析器结构，但权衡之下，比起提高分析器能力的好处，增加一点复杂度还是值得的。

LL(*k*)分析器到底能比LL(1)强多少？增大前瞻集，使LL(*k*)分析器能解析更多的计算机语言。但在消除语法规则的二义性方面，它仍然只能应付*k*个符号以下的情况。这时可以为LL(*k*)分析器增加另外一些巧妙的特性。*回溯*即为其中之一，具备回溯能力的分析器可以识别具有任意前瞻集的语言。7.3.2节将讨论回溯分析器。

ANTLR可生成能处理任意前瞻集的LL(*k*)分析器，最适合在实际应用中实现DSL。Google App Engine、Yahoo Query Language（YQL）、IntelliJ IDEA等许多大型软件应用都采用了ANTLR来分

析、解释其自定义语言。

掌握了分析器的基本形式，下面开始讨论一些比较高级的自顶向下分析器。这些分析器的使用频率可能不高，但还是值得我们了解其实现，学习它们为提高效率而采取的技术手段。何况第8章讨论通过Scala分析器组合子设计外部DSL时，将要用到其中一种技术。

7.3.2　高级的自顶向下语法分析器

高级的语法分析技术可以赋予分析器更强大的功能，代价是增加实现上的复杂性。不过一般我们通过分析器生成器、分析器组合子等抽象手段来间接实现分析器，其实现复杂性已经被封装在抽象内部。开发者使用的只是抽象对外公开的接口而已。

1. 递归下降回溯分析器

这种分析器在LL(*k*) RD分析器的基础上增加了回溯机制，因此能够处理任意大小的前瞻集。在回溯机制的帮助下，分析器可以根据需要向前试探。如果找不到匹配项，分析器则回滚其输入，换成别的规则重新进行尝试。与LL(*k*)分析器相比，这种试探机制使回溯分析器的分析能力有了极大提升。

分析器在回溯并选择另一条规则时，有没有选择的优先次序呢？以ANTLR为例，我们可以通过语法谓词（syntactic predicate）的形式，向分析器提示规则的优先次序（见7.6节文献[2]）。分析器根据我们指定的顺序，选择最恰当的规则用于输入流。

这类分析器支持一种表达能力更强的语法形式，称为PEG（parsing expression grammar，解析表达语法）。PEG对ANTLR的回溯和语法谓词进行了扩展，提高了表现力。它加入了`&`、`!`等运算符用于文法规则的定义，可对回溯和分析器的行为进行更细致的控制。文法描述本身读起来也好理解得多。运用中间结果记忆（memoizing）等技术，我们可以开发出线性时间的PEG分析器。

2. 带中间结果记忆的分析器

回溯RD分析器在执行回溯，尝试其他规则时，常常要重复计算一些分析结果。带记忆的分析器通过缓存分析的部分中间结果，提高了分析的效率。

提高效率很好，不过加入记忆机制，意味着需要更多的内存空间来放置前面的计算结果。传统回溯分析器实现的效率大为提高，增加一套机制所带来的麻烦还是值得的。况且，我们的老朋友ANTLR支持记忆功能，并不需要我们亲自动手。

记忆分析器需要占用更大内存空间的弱点，可以通过一种叫做*Packrat分析器*的实现方案来规避。这种分析器除了它奇特的名字，更为引人注目的是其函数式特征（详见7.6节文献[6]）。Haskell等函数式语言具备的惰性特性可以很自然地应用在Packrat分析器的实现当中。8.2.3节谈及各种Scala分析器时，会再次详细探讨Packrat分析器。

3. 语义谓词分析器

有时，RD分析器无法仅凭语法本身判断应该应用的规则。我们可以在分析器上标注一些Boolean表达式，以帮助它做出决定。只有当Boolean表达式求值为真时，候选规则才算匹配成功。

语义谓词分析器的典型用途是，用单个分析器来处理一种语言的多个版本。分析器的基干承担核心语言的识别工作，而语言的扩展和其他版本，则由附加的语义谓词去解决。对语义谓词分

析器的详细解说请参阅7.6节文献[1]。

自顶向下分析器非常简单，类似于7.3.1节的LL(1)。但对于复杂的语言分析，我们需要准备能力与之相称的分析器，例如本节介绍的回溯分析器、记忆分析器、语义谓词分析器。这些高级分析技术适应的语言结构范围更广，其实现的时间复杂度和空间复杂度有所降低。下一节将介绍另一类分析器，它们可以识别任意的确定性上下文无关语法，从这个意义上说，他们比自顶向下分析器适用性更强。

7.3.3　自底向上语法分析器

自底向上分析器从叶子开始构造分析树，逐步移向根节点。分析器从左向右扫描输入流，通过对文法规则的后续规约构造最右推导，朝语法的起始符号方向处理。方向与自顶向下分析器正好相反。

自底向上分析器最为常用的实现技法称为*移进-归约*（shift-reduce）分析。分析器扫描输入并遇到一个符号时，它有两种选择。

- 将当前符号*移进*到一旁（通常推入某种符号栈）供后续归约。
- 匹配当前*句柄*（handle）（即输入串中，与一条产生式规则右部相匹配的子串），并以产生式规则左部的非终结符号替换该句柄。这个步骤一般称为“归约”。

我们将介绍两种最为常用的移进-归约式自底向上分析器，一种是算符优先（operator precedence）分析器，另一种是LR分析器。算符优先分析器功能有限，但手工实现起来极为简单。相对来说，LR分析器在各种生成器中得到了非常广泛的应用。

1. 算符优先分析器

这种自底向上分析器只能识别数量有限的语言类型。它基于分配给各终结符号的一组静态优先关系规则。

算符优先分析器很容易手工实现，但由于它既处理不了同一符号的多重优先关系，又不不能识别存在并列的非终结符号的文法，因而适用范围受到限制。例如下面的片段就不符合算符优先文法的要求，因为规则`expr operator expr`中含有多个相邻的非终结符号。

```
expr : expr operator expr
operator : + | - | * | /
```

接下来我们要介绍一种应用最为广泛的自底向上分析器，它能实现非常多的语言类型，也为YACC和Bison等流行的分析器生成器所采用。

2. LR(*k*)分析器

LR(*k*)分析器是效率最高的自底向上分析器，它可以识别一大类上下文无关文法。LR(*k*)名字中的L指分析器从左向右扫描输入串。R指其分析过程是构造最右推导的逆过程。最后的*k*显然还是指前瞻符号的数目，分析器依据*k*个前瞻符号来决定适用的产生式规则。

LR分析器由分析表驱动。生成器将分析器识别出输入符号时应该执行的操作，存储在一张分析表中。这里所谓的操作其实就是*移进*或者*归约*。整个输入串识别完毕，我们说成分析器“归约到了文法的起始符号”。

这种类型的分析器很难手工实现，但YACC、Bison等生成器对LR分析器的支持十分完善。

3. LR分析器的变体

LR分析器有三种变体：简单LR（SLR）、前瞻LR（LALR）、规范LR（canonical LR）。SLR分析器使用简单的逻辑来判断前瞻集合，分析过程容易产生大量的冲突状态。LALR分析器较SLA复杂，对前瞻的处理更周详，冲突也更少。规范LR分析器比LALR能识别更多类型的语言。

4. 我们真正会用的方法

分析器是外部DSL的核心所在。我们需要基本了解分析器与被识别的语言类型之间的关系。本节介绍了很多这方面的专门知识，应该能帮助选择正确类型的分析器去实现DSL。不过在现实中，除非要实现的语言实在太过简单，否则我们绝对不会手工实现分析器。使用分析器生成器才是正确选择。

使用生成器，我们可以在更高级的抽象上思考。定好文法规则（也就是语言的语法），然后生成器帮助构建实际的分析器实现。不过别忘了，生成的实现中只包含语言的识别机制和一棵简单的AST树。我们还要进一步将AST转换为语义模型，才能满足DSL处理的实际需求。建立语义模型的方法是在文法规则中嵌入自定义的操作代码。

下一节要在DSL开发过程里导入丰富的工具支持，从而在更高级的抽象上使用生成器进行DSL开发。

7.4　工具支持下的 DSL 开发——Xtext

像ANTLR这样的语法分析器生成器，对于在更高级的抽象上开发外部DSL是一大进步。不过我们还是需要直接在文法规则中嵌入目标操作。EBNF规则没有与语义模型的生成逻辑充分分离。假如我们希望为一套文法规则实现多个语义模型，除了复制分析器本身的代码，或者通过文法规则提供的共同抽象基础去派生不同的语义模型，恐怕没有更好的办法。

Xtext是一个建立在Eclipse平台基础上的外部DSL开发框架。它提供了管理DSL完整生命周期的全套工具。Xtext集成了EMF（Eclipse Modeling Framework），并且会以组件形式管理DSL开发过程中产生的所有内容。（关于EMF框架的详情请查阅http://www.eclipse.org/emf。）

使用Xtext时，我们首先编写好语言的EBNF文法。然后EMF根据文法生成以下产物：

- 基于ANTLR生成的语法分析器；
- 语言的元模型，基于EMF的Ecore元模型语言；
- 一个自定义Eclipse编辑器，带有语法高亮、代码提示、代码补全功能，还为我们的模型提供一个可定制的大纲视图。

图7-9展示Xtext对我们定义的EBNF文法规则做了哪些后续处理。

Xtext还拥有各种可以组合使用的代码生成器，能生成满足多种需求的语义模型。本节将再次实现7.2.2节用ANTLR实现过的交易指令处理DSL。这次你会注意到，Xtext全方位的工具支持和它基于模型的开发方式，大大方便了我们管理DSL的演化。我们的实现历程将从语言的定义开始，使用一种基于EBNF的Xtext文法规则。

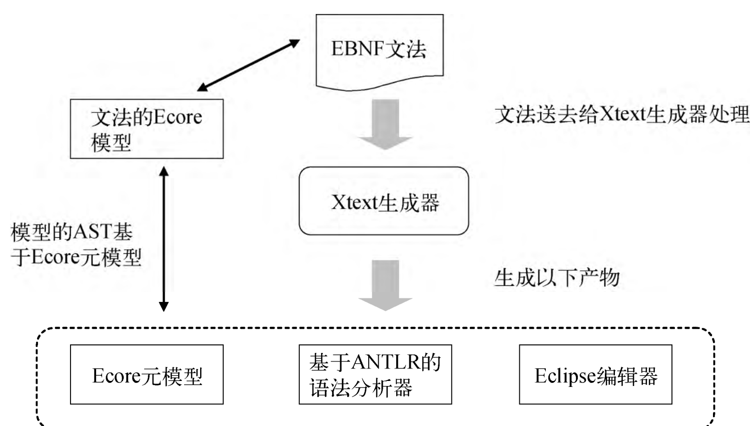


图7-9　Xtext处理文本性质的文法规则过后将生成大量产物。其中Ecore元模型抽象了文法模型使用的语法，是众多制成品里的重中之重

7.4.1　文法规则和大纲视图

Xtext的DSL文法定义沿用EBNF形式，另外有一些Xtext的附加功能点缀其间。这里不打算详细介绍Xtext的文法规则定义，这方面的信息都可以在*Xtext User Guide*（见7.6节文献[5]）里面查到。代码清单7.6用Xtext文法规则重新定义了7.2.2节的交易指令处理DSL。

代码清单7-6　Xtext文法规则

```
grammar org.xtext.example.Orders
  with org.eclipse.xtext.common.Terminals

generate orders http://www.xtext.org/example/Orders

Model :
  (orders += Order)*;

Order :
  line = Line;

Line :
  buysell = ('buy' | 'sell') security = Security price = Price account = Account;
Security :
  name = ID;

Price :
  '@' (
        (value = INT)
        |
        ('limitprice' '=' (value = INT))
      );

Account :
  'for' value = ID;
```

❶ 重用默认的词法分析
❷ 生成元模型器
❸ 多值赋值
❹ 单值赋值

7

这段代码与先前的ANTLR版本大体相似。其中一处不同点表现在开头部分，即命令Xtext生成语言元模型的部分❷。假如已经有现成的Ecore元模型，也可以让Xtext将它导入当前工作环境，令模型与文法规则的文本表述同步。7.4.2节会进一步探讨元模型的内部细节。Xtext允许将既有文法混入当前正在定义的规则❶，以此实现文法的重用，这也是一个非常有趣的特点。

Xtext根据我们的文法规则生成默认的分析树（AST）。AST生成之后，文法里的内联赋值成为语法分析器生成的各AST元素之间的联系纽带❸、❹。

除文本性的文法规则之外，Xtext还给出一个大纲视图来展现模型的树形结构。利用大纲视图，可以浏览各模型元素的。图7-10显示根据代码清单7-6定义的文法产生的大纲视图。

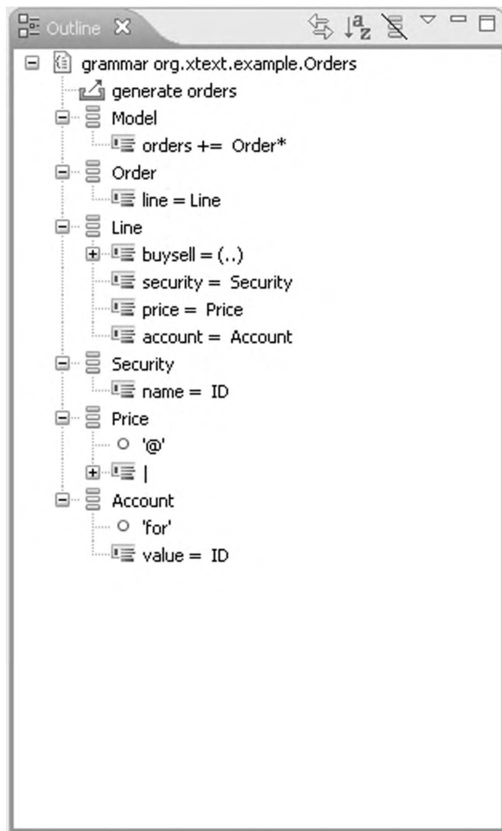


图7-10　大纲视图展示模型的层级结构。大纲视图展示每一条规则对应的结构。视图内的元素可以按字母顺序排列，以便查找，选择其中一个元素将打开相应的文本编辑器

图中所见即为模型的默认视图。这个视图最值得称道的特点在于，Xtext允许定制视图的几乎每一个方面。我们可以调整大纲的结构，可以让用户选择过滤部分内容，可以自定义上下文菜单等，只要覆盖默认实现即可。大纲视图是实现语言模型可视化的其中一件工具，与文法的文本表述结合起来，赋予DSL开发过程更加丰富的体验。

7.4.2　文法的元模型

在文本编辑器里写好文法规则以后，让Xtext生成语言制品，它就会根据文法生成完整的Ecore元模型（Ecore包含EMF的核心抽象定义）。我们以文本形式定义的文法，经过元模型的抽象，呈现为一种便于Xtext管理的模型样式。元模型用Ecore元类型（metatype）来描述文法规则的各个部分。图7-11展示了Xtext文法示例所对应的元模型。

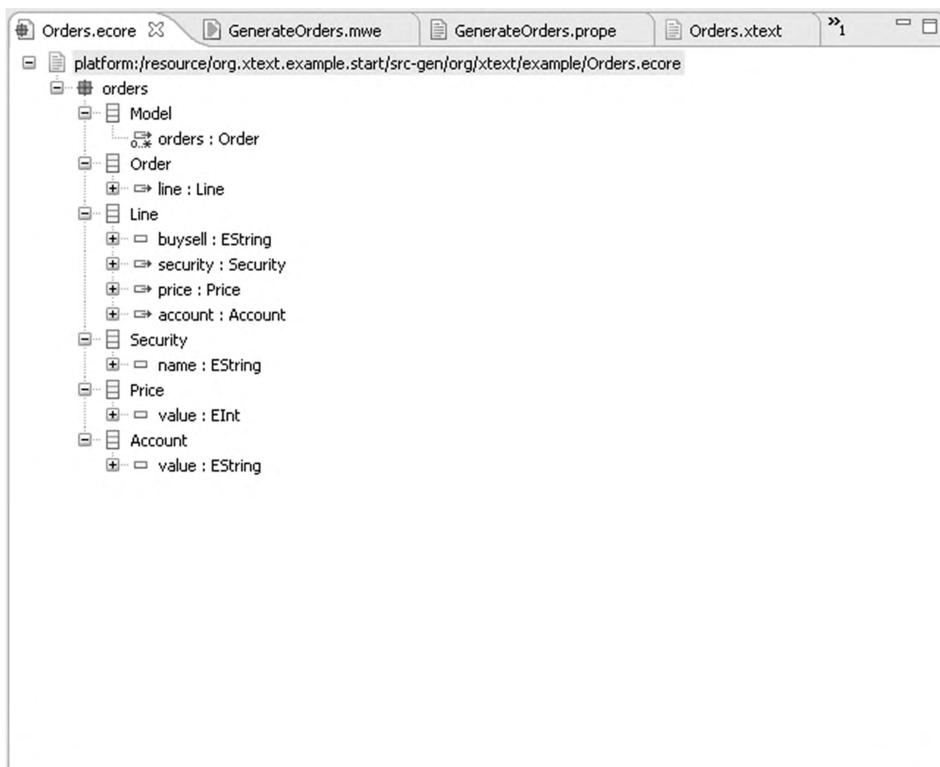


图7-11　交易指令处理DSL的元模型。文法中的每条产生式规则都返回一个Ecore模型元素，如`EString`和`EInt`

注意，在这个元模型里用了`EString`、`EInt`等元类型来表示文法的`AST`，它们都是Ecore提供的核心抽象。

除了元模型，生成器同时还生成用来实例化元模型的ANTLR分析器。元模型控制Xtext提供的全套工具。如需进一步了解元模型的内部细节，请参考*Xtext User Guide*（参考7.6节文献[5]）。

所有必要的制品都生成完毕，连同生成的IDE插件也都安装好之后，就可以在编辑器里编写DSL脚本，享受语法高亮、代码提示、约束检查等智能功能。Xtext在它的信息库中建立了DSL语言完整的EMF模型，因而可以在脚本的呈现方式上增加智能化的手段。图7-12展示了DSL示例的编辑场景。

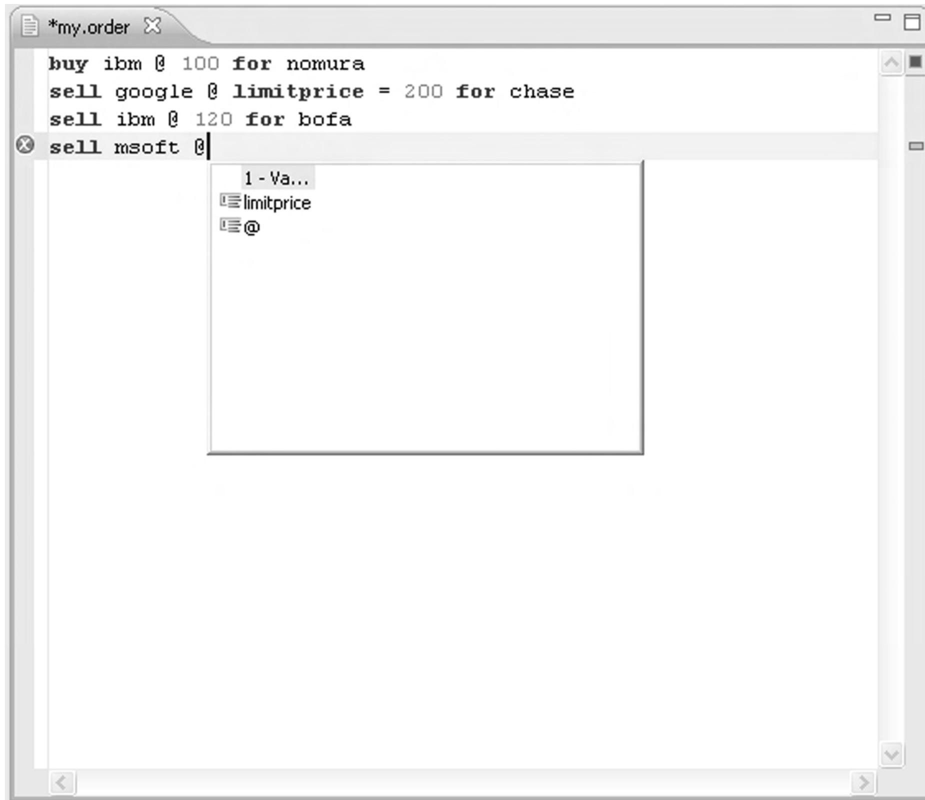


图7-12　Xtext元模型提供了一个专为编写DSL而设的优秀编辑器。图中代码补全功能提示有效的输入候选内容。从图中还可以看到另一种便利功能——语法高亮功能

现在，我们已经将语言签入了Xtext信息库中。Xtext为我们提供操作DSL语法的手段和界面，并且自动更新其Ecore模型。它还给了我们一棵默认的AST树，以及生成这棵AST的语法分析器。但对于一种实用的DSL来说，这些还不够，我们还需要一个更加精细、完善的抽象——语义模型。DSL设计者希望通过自定义的代码开发来产生语义模型，同时希望这部分代码能与语言的核心模型分离。Xtext的代码生成模板提供了这样的能力，可以妥善地将生成的语义模型与文法规则集成在一起。那么，下面就让我们继续探索Xtext这方面的功能，看看它的代码生成器要用哪些工具来为语义模型生成代码。

7.4.3　为语义模型生成代码

文法规则和元模型都准备就绪，可以编写代码生成器了。代码生成器将处理我们到目前为止所建立的模型，并生成语义模型。有时，我会希望从一套文法生成多个语义模型。以交易指令处理DSL为例，除了生成一个根据用户输入创建交易指令对象集合的Java类，我们可能还希望生成一个JSON（JavaScript Serialized Object Notation）对象集合，用来向数据库传递交易指令数据。

理想状态下，这两个语义模型都不与核心的文法规则发生耦合。图7-13展示了整体架构。

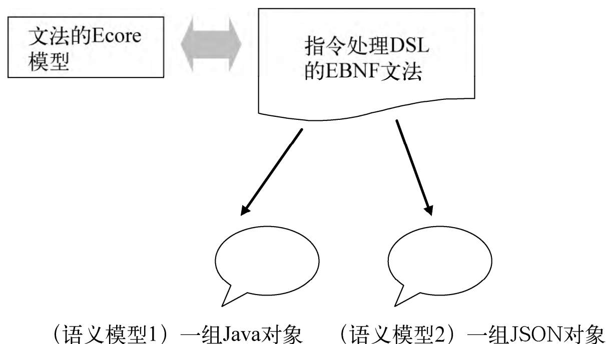


图7-13　语义模型应与文法规则分离。一套文法规则可以对应多个语义模型

那么，我们就用Xtext提供的Xpand模板来生成Java代码。

1. 用Xpand模板生成代码

Xpand模板遍历由文法规则形成的AST，然后生成代码。模板需要的第一个输入是元模型，我们通过«IMPORT orders»提供给它。下面是充当入口的主模板，它负责派发到其他子模板：

```
«IMPORT orders»
«DEFINE main FOR Model»
  «EXPAND Orders::orders FOR this»
«ENDDEFINE»
```

在这段代码中，orders是需要导入的元模型名称。每个模板都有一个名称和一个决定模板调用时机的元类型。本例中模板的名称是main，元类型是Model（Model是我们在文法规则中定义的一个文法元素，Xtext将它表示为Ecore元模型中的一个元类型）。这个main模板非常简单，它的功能仅仅是在识别到Model符号时，派发到Orders::orders子模板。代码清单7-7是Orders::orders模板的定义。

代码清单7-7　生成orders语义模型的模版

```
«IMPORT orders»                                          <-- 导入元模型

«DEFINE orders FOR Model»
  «FILE "OrderProcessor.java"»                           <-- 生成Java文件
    import java.util.*;
    import org.xtext.example.ClientOrder;                <-- 自定义Java类
    public class OrderProcessor {                        <-- 待生成的类
      private List<ClientOrder> cos = new ArrayList<ClientOrder>();
      public List<ClientOrder> getOrders() {
        «EXPAND order FOREACH this.orders»               <-- 将被替换的模板
        return Collections.unmodifiableList(cos);
      }
    }
  «ENDFILE»
«ENDDEFINE»
```

7

```
«DEFINE order FOR Order»
        cos.add(new ClientOrder("«this.line.buysell»",
          "«this.line.security.name»",
          «this.line.price.value»,
          "«this.line.account.value»"));
«ENDDEFINE»
```

当文法被归约到起始符号（即`Model`），语言识别结束时，代码清单7-7中定义的代码也在模板推动下生成出来。表7-4就代码生成模板的一部分特性给出了详细解释。

表7-4　Xtext的代码生成模版

特　　性	解　　释
可以生成任意的代码。例中生成了一个Java类	这个类只对外提供一个方法，返回从DSL脚本解析所得的全部交易指令的集合。集合的元素是单独定义的POJO对象（`ClientOrder`），与文法无关
在`orders`模板内运用了对`order`模板的内联展开	我们可以在产生式规则内访问文法模型的元素，并使用模板替换这些内联元素的文本。例如`«this.line.price.value»`就是一个占位标记，会在DSL脚本完成分析后，被实际的单价替换掉

模板一切就绪，我们通过项目的上下文菜单再次运行Xtext的生成器。

2. 执行DSL脚本

假设我们要执行下列DSL脚本：

```
buy ibm @ 100 for nomura
sell google @ limitprice = 200 for chase
```

这段脚本经过Xtext的处理，最终生成代码清单7-8所示的OrderProcessor.java文件。

代码清单7-8　由代码清单7-7的模版生成的类

```
import java.util.*;
import org.xtext.example.ClientOrder;
public class OrderProcessor {
  private List<ClientOrder> cos = new ArrayList<ClientOrder>();

  public List<ClientOrder> getOrders() {
    cos.add(new ClientOrder("buy", "ibm", 100, "nomura"))
    cos.add(new ClientOrder("sell", "google", 200, "chase"))
    return Collections.unmodifiableList(cos);
  }
}
```

按照Xtext的做法，定义文法规则和构建语义模型这两个方面可以充分分离。文法规则的定义由智能文本编辑器负责，用可定制的大纲视图作为补充。语义模型的实现则在各种代码生成器的控制之下，通过元模型与语法分析器联系在一起。

最后谈谈开发外部DSL时，像Xtext这样文本与可视化环境相混合的方式有哪些优缺点。

3. 优点和缺点（优点是主要的）

Xtext提出了一种创新的外部DSL开发方式，以其丰富的工具，增强了DSL传统的文本表示。我们仍然需要编写EBNF格式的文法规则，但在后台，Xtext不但为我们生成ANTLR语法分析器，

更将我们的文法规则抽象为一个元模型。 Xtext的全套工具都置于一个围绕元模型建立起来的架构内。基于Xtext的开发方式除了增强DSL编辑能力，用它的Xpand模板机制来生成自定义代码，还能有效地分离语义模型和文法规则。

总而言之，在EMF框架支持下，Xtext提供了优异的外部DSL开发体验。只有一点需要小心，即Xtext对Eclipse平台的依赖。不过也只有IDE集成开发环境依赖Eclipse，语法分析器、元模型、序列化组件、链接组件、Xpand模板等运行时组件，都可以在任意的Java进程中使用。Xtext的种种优点使其成为我们开发外部DSL的首选框架。

7.5 小结

本章学习了外部DSL的设计原则。外部DSL需要建立自己的一套语言处理设施。假如我们的DSL复杂度特别低，那么可以手工编写语法分析器。复杂一些的DSL则需要用到功能完备的语法分析器生成器，如YACC、Bison、ANTLR等。我们详细讨论了基于ANTLR的语言构建，用它开发了一种自定义交易指令处理DSL。其间考察了ANTLR实现引擎中的各组成部件，如词法分析器、语法分析器、语义模型等，如何协作形成最终的DSL处理程序。

要点与最佳实践

- **设计DSL时**，应该明确分离**语法和背后的语义模型**。
- 外部DSL的**语义模型可以用宿主语言的语言结构来实现**。语法分析需要一种能与宿主语言集成的分析器生成器。ANTLR是一款典型的分析器生成器，它能和Java语言完美地集成在一起。
- **选择合适的分析器类型**。只要语言不是太过简单，都应该在开始外部DSL设计之前，就按照对语言处理能力的需求做好决策。正确的分析器类型有利于降低实现的复杂度。
- **按实际需要来决定复杂度水平**。外部DSL不必设计得像通用语言一样复杂。

对于具备一定复杂度，语法比较丰富的外部DSL来说，语法分析器是语言设计的核心。语法分析器按照其扫描输入流和构造分析树的方式来分类。主要有两大类型：自顶向下分析器和自底向上分析器。语言设计者需要知道每一类语法分析器适合处理的语言种类，还需要知道每一种分析器实现的复杂度和选择依据。本章一边介绍语法分析器的分类，一边深入讨论了这方面的内容。

完成交易指令处理DSL的设计，说明我们已经懂得为语言选择正确的分析器类型。本章最后一节转向另一种DSL开发范式，在熟识的标准文本模型基础上，引入一套丰富的工具。立足Eclipse平台，再结合EMF框架的模型驱动开发方式，令Xtext成为集优点于一身的外部DSL开发环境。用Xtext重新实现与前面的例子相同的交易指令处理DSL。开发过程体现了模型驱动方式的多样性结合一整套工具，可以获得更加丰富的语言开发体验。

下一章将介绍使用Scala语言的一种截然不同的外部DSL开发范式。Scala提供的一类函数式组合子可以作为开发语言分析功能的建造材料。它们称为分析器组合子（parser combinator）。我们将用这些组合子来实现证券交易领域的外部DSL示例。

7.6　参考文献

[1] Parr, Terence. 2009. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. The Pragmatic Bookshelf.

[2] Parr, Terence. 2007. *The Definitive ANTLR reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf.

[3] Aho, Alfred V., Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools*, Second Edition. Addison Wesley.

[4] Ford, Bryan. 2004. Parsing Expression Grammars: A Recognition Based Syntactic Foundation. *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pp 111-122.

[5] *Xtext User Guide*. http://www.eclipse.org/Xtext/documentation/latest/xtext.html.

[6] Ford, Bryan. 2002. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pp 36-47.

第 8 章

用Scala语法分析器组合子设计外部DSL

8

本章内容

- 什么是分析器组合子
- Scala的分析器组合子库
- Packrat分析器的用法
- 用Scala提供的各种分析器组合子来设计外部DSL

带着我们从第7章学到的关于外部DSL实现的基础知识，本章把话题直接转到分析器组合子（parser combinator）。分析器组合子是函数式编程最为精彩的应用之一。这些组合子构成了一种用来设计外部DSL的内部DSL，也就是说，可以不必像别的外部DSL实现技术那样，要求我们自行建立一套语言处理设施。以第7章处理客户交易指令的外部DSL为例，我们在设计的时候用到了ANTLR语法分析器生成器。设计好的DSL语法要通过ANTLR来生成语法分析器。换言之，我们的DSL需要依赖外部工具来提供必要的语言实现基础设施。而当我们使用分析器组合子的时候，完全不需要越出宿主语言半步。实现因此变得简洁、有表现力，而且完全摆脱了一切外部依赖。

我们先从什么是分析器组合子说起，再谈到Scala在其核心语言基础上实现的分析器组合子库。随后进一步详尽地说明Scala组合子库的各种细节，重点突出那些令其成为DSL设计标杆的特性。在这之后，我们将运用Scala的分析器组合子来设计两个外部DSL。最后介绍packrat分析器，这种分析器能实现普通递归下降分析器无法实现的文法类型。图8-1是本章的路线图，我们就照着图上的指引，循序渐进地探索用Scala的分析器组合子来设计DSL的世界。

学习完本章内容，你将牢固地掌握如何使用函数式编程技术，特别是分析器组合子技术来实现可扩展的外部DSL。

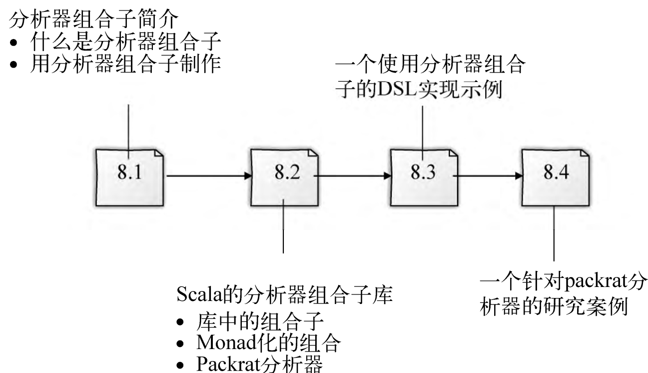


图8-1　本章路线图

8.1　分析器组合子

在第7章的时候，我们把语法分析器定义成一个作用于输入流，并将词法单元集合的引擎。它能在符号流中识别分析器认可的有效语言成分，或者在遇到无效符号的时候立即中止当前输入。无论哪种情况，分析器都返回一个（或成功或失败的）结果，同时返回尚未处理的剩余输入流。

如果分析器返回成功的结果，我们可以将剩余输入流送入另一个分析器继续处理。如果返回失败结果，我们可以回退到输入流的开头位置，尝试用另一个分析器来处理它。鉴于分析器自身的工作模式，可以将多个分析器按不同方式串联起来，实现对输入流的完整分析。图8-2描绘了将多个分析器组合起来，共同处理输入流的情形。

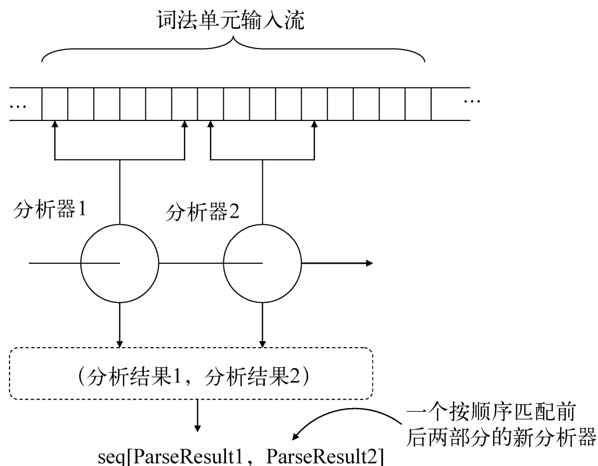


图8-2　串联起来的分析器。分析器1处理一部分输入流，分析器2处理分析器1余下的部分。这两个分析器组合而成的分析器，其返回结果也是原来两个返回结果的组合。只有当分析器1和分析器2都成功匹配其输入时，组合后的分析器才返回成功结果

本节将建立一种分析器的使用思路，即分析器是对它所识别的语言的一种函数式抽象。由于分析器是其输入的函数，所以可以一个一个地组合成别的分析器，而且每次组合都是DSL语法的一次扩增。对于能完成这样的组合的高阶函数，我们赋予它一个正式的名称：分析器组合子。

8.1.1　什么是分析器组合子

我们以函数式的思路来考虑分析器的组合问题。在函数式编程中，分析器是一个接受输入并产生结果的函数。分析器组合子允许我们单纯以组合的方式，将高阶函数（又叫做组合子）搭建成各式文法结构，如顺序、重复、可选项、分支选择等。如果宿主语言支持中缀运算符写法，那么用分析器组合子写出来的文法规则，看起来就像EBNF产生式的样子。

用组合子来分析语法，最大的好处在于能够提高组合性；少数基本的分析器经过函数式的组合，即可构成更大更复杂的分析器（本书附录A阐述了组合性特质的优点）。组合子的编排就像搭积木，我们从小块的材料开始，逐渐搭出高阶的结构。图8-2就是一个顺序组合子在履行它的搭建工作。

把搭积木的思路推广到DSL设计上，我们先用分析器组合子组合出一些小的语言片段，作为对DSL语法局部的建模，再由这些片段拼出完整的DSL结构。图8-2的顺序组合子只是众多组合子中的一种，图上它正在将两个DSL语法分析器顺序地连接起来。任何组合子库都会包含各式各样的组合子，就像一套积木里有不同的形状。我们从几个常用的组合子说起，看它们怎样处理输入流和识别语言文法，详见表8-1。

表8-1　常用的分析器组合子

组　合　子	组合方式
顺序	用于构造顺序结构的分析器组合子。若分析器P和Q以顺序组合子相连接，则当出现以下情况时，认为分析是成功的 ■ P成功处理一部分输入流 ■ Q跟在P之后处理P未处理的剩余输入
替代	用于构造替代结构的分析器组合子。若分析器P和Q以替代组合子相连接，则当P或Q其中之一在以下情况下成功时，认为分析是成功的 ■ P先处理输入流。若P成功，则分析是成功的 ■ 若P失败，则输入流回退到P开始处理之前的位置，换由Q来处理同一输入流 ■ 若Q成功，则分析是成功的；否则分析失败
函数施用	这个组合子在分析器上应用一个函数，结果产生一个新的分析器
重复	当重复组合子作用于分析器P时，返回另一个分析器，其分析对象是P的分析对象的一次或多次重复。有时候，重复组合子还允许重复模式与分隔符交错的情况 例如，P可分析字符串abc，对P应用重复组合子后产生的分析器，将能够分析abc的重复abcacabc…，或者允许模式abc与空格交错出现，即abc abc abc…

仅仅知道这几个基本的组合子，还不够用于讨论DSL的具体实现。我们要先学习怎样用分析器组合子来设计外部DSL。

8.1.2　按照分析器组合子的方式设计DSL

第7章我们在设计外部DSL时，自行建立了一套语言处理设施。由于手工实现分析器工作量巨大又容易出错，代码还常常膨胀到失控的地步，所以我们决定借助ANTLR和Xtext等外部框架。在外部框架的参与下，会形成如图8-3所示的实现架构。

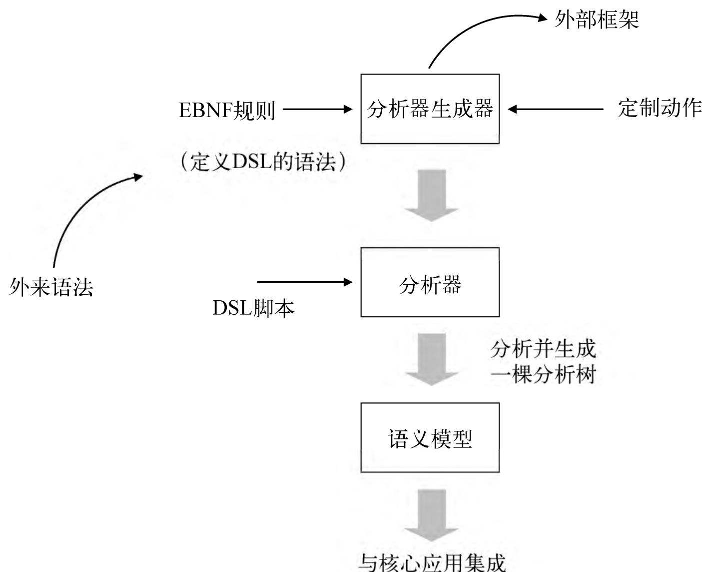


图8-3　使用ANTLR等外部分析器生成器来设计外部DSL的实现架构。生成器产生分析器，分析器分析DSL脚本并生成应用的语义模型

图上的架构并不是一个坏的架构，相反，它是当前使用最为普遍的外部DSL设计范式。自从LEX、YACC那一代语言处理工具走出AT&T实验室以来，开发者们一直在沿用相同的架构风格。

虽然这个架构久经考验，但并不意味着我们不能通过探索找到更新更好的DSL实现方式。图8-3的架构有一个明显的缺点，就是这样实现出来的DSL具有外部依赖。分析器生成器作为一个外部实体（如图中所见），我们需要用它不同于宿主语言的语法来定义DSL的EBNF规则（请回忆一下第7章的EBNF知识），这就使得学习曲线更为陡峭。生成器生成的分析器代码结构是静态的，完全依赖于生成器内部的实现，用户没有多少调整定制的余地。

用分析器组合子来设计DSL是一种完全不同的体验。我们可以沉浸在宿主语言的抽象氛围里定义文法规则，用高阶函数定义DSL的语法，用库里预备的组合子添加定制动作。具体的细节我们会在后续章节内详细解说。从图8-4可以窥见宿主语言内生的分析器组合子对简化实现架构的成效。

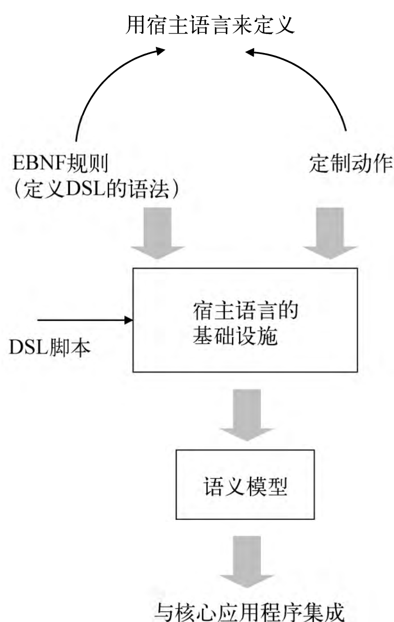


图8-4　使用分析器组合子来设计外部DSL的实现架构。定义文法规则和定制动作时，完全不需要越出宿主语言的设施范围之外

在新的架构下，DSL不存在对外部框架的依赖。仅有的先决条件，是宿主语言必须提供一套分析器组合子库。相对而言，分析器组合子是DSL实现领域的新成员。不少现代语言，如Haskell、Scala和 Newspeak都在其核心语言上，以库的形式提供分析器组合子功能。你将在本章的学习过程中发现，分析器组合子蕴含了对函数式编程思维精彩而新颖的运用。我们要揭开它是怎么做到设计DSL时简洁而不失表现力的。

定义　Newspeak是由 Gilad Bracha设计的一种沿袭Self和Smalltalk语言传统的编程语言。关于这种语言的详情请参阅http://newspeaklanguage.org

---

下一节，我们将认识Scala的分析器组合子库，这个库以纯函数式的方式提供强大的外部DSL设计能力。我们定义的每一个分析器都是对DSL语法的一个小成分的建模，组合子像胶水一样把所有的成分连接起来，赋予它们语义。

8.2　Scala 的分析器组合子库

Scala在其核心语言之上实现了分析器组合子库。这个库随Scala语言一起发布，包的位置在`scala.util.parsing`。以库的形式实现分析器组合子，便于在不影响核心语言的前提下进行扩展。本节将通过各种DSL片段来学习Scala库的使用技巧和惯用法。API方面的细节可以参考8.6

节文献[1]和文献[2]，或者直接查阅Scala的源代码。（不幸的是，目前还没有详细介绍Scala分析器组合子的出版物，源代码是眼下最好的参考资料。）

8.2.1　分析器组合子库中的基本抽象

通过上一节的讨论，我们知道分析器是一个将输入流变换为分析结果的函数。Scala库根据这个概念建立了如图8-5所示的模型。

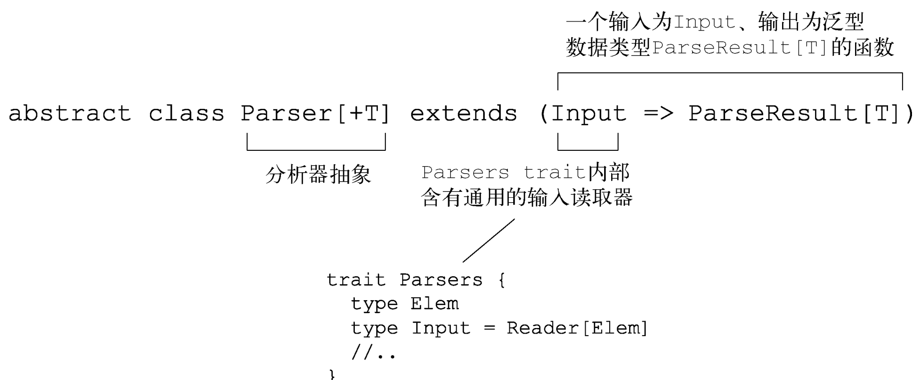


图8-5　Scala库将分析器建模为一个函数

`ParseResult`是对分析器产生的结果的抽象，结果可以是成功也可以是失败。此外`ParseResult`还跟踪着尚未被当前分析器处理的下一输入。Scala库建模的`ParseResult`是一个泛型抽象类，`Success`和`Failure`是它的两个特化实现。下面的代码清单给出了Scala对`ParseResult[T]`类型的定义，以及该类型的特化实现`Success`和`Failure`。

代码清单8-1　Scala对分析结果的建模

```
trait Parsers {
  sealed abstract class ParseResult[+T] {
    //..
    val next: Input                          ❶ ParseResult跟踪着下一输入
  }

  case class Success[+T](result: T, override val next: Input)
    extends ParseResult[T] {                 ❷ 分析成功
    //.. implementation
  }

  sealed abstract class NoSuccess(
    val msg: String, override val next: Input)
      extends ParseResult[Nothing] {         ❸ 针对分析不成功情况的基类
    //..
  }

  case class Failure(
    override val msg: String, override val next: Input)
```

```
        extends NoSuccess(msg, next) {          ◁── ❹ 失败 => 回溯并重试
      //..
    }
    case class Error(
      override val msg: String, override val next: Input)
        extends NoSuccess(msg, next) {          ◁── ❺ 不可恢复的错误，不回溯
      //..
    }
    //..
  }
```

ParseResult对分析器产生的结果的数据类型做了泛型化处理。当结果为Success ❷时，我们得到类型为T的结果。当结果为Failure ❸时，我们得到一条失败消息。Failure分为可恢复错误（nonfatal）和不可恢复错误（fatal）。对于可恢复的Failure ❹，我们可以回溯并尝试其他备选的分析器。对于不可恢复的Error ❺，不存在任何回溯，分析过程终止。不管发生哪种情况（Success或Failure），结果都会说明当前分析过程处理了多少输入，分析器链条中的下一个分析器应该从输入流的哪个位置开始接手❶。

本章后续的外部DSL例子会继续演示替代和回溯的具体做法。如何处理分析中出现的不成功的情况，这是我们设计DSL时必须仔细考虑的一个要点。例如有时候替代安排得太多，可能导致性能衰退，而且有时候客观条件不允许我们设计带回溯的分析器。



你想知道分析器组合子库里的这些类都在DSL实现中扮演什么角色吗？

我们建模时每一段DSL都要有一个配套的分析器，由它负责检查语法的有效性。只有当用户提供的脚本语法正确时，DSL处理过程才会继续下去。如果语法是有效的，分析器返回Success；否则返回Error或Failure。对于Failure的情况，分析器可以进行回溯，尝试用别的替代规则来分析。

现在我们知道了分析器和ParseResult的各种实现，但那么多分析器是怎么串联在一起，完成整个 DSL的分析工作的呢？这正是组合子的作用。所以，接下来我们要介绍Scala提供的一些组合子，学习怎样高效率地利用这些组合子把我们设计的分析器一个个连接起来。

8.2.2 把分析器连接起来的组合子

Scala分析器组合子库含有一整套用来连接各种分析器的组合子。我们在第7章用ANTLR和Xtext来设计外部DSL的时候，利用分析器生成器技术同样可以写出近似EBNF形式的文法。组合子技术与之相比，不需要借助宿主语言以外的任何外部环境。我们在Scala语言范围内，利用其高阶函数特性即可定义出类似EBNF的文法。假如把DSL语法看作是众多小片段的集合，那么组合子的作用就是把每个片段对应的分析器拼接到一起，构成一个大的、针对DSL整体的分析器。

学习Scala组合子的最佳方式自然是通过真实的例子。我们继续沿用前面章节用过的领域示例，设计一种外部DSL来处理用户通过一系列输入提供的客户交易指令。我们需要生成的抽象如图8-6所示。

8

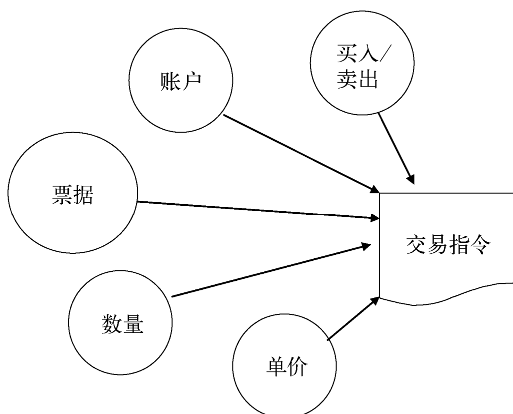


图8-6　以各种输入项目作为属性来生成交易指令

大致掌握分析器组合子的能力之后，我们现在多了一种可行的选择。排除了通过外部分析器生成器（如ANTLR）的实现方式，我们还可以考虑用Scala提供的组合子把一系列较小的分析器连接起来，实现对DSL的语法分析。

分析链条中的每一个小分析器只负责分析一小块固定的DSL结构，然后在组合子的协调下，将输入流转交给下一个分析器。经过与客户的反复商讨和迭代开发，我们确定了客户希望的语法，最后得到下面代码清单中的文法定义。这段代码使用了我们刚刚介绍过的Scala分析器组合子来表达DSL语法。

代码清单8-2　用Scala分析器组合子设计外部DSL的示例

```
package trading.dsl
import scala.util.parsing.combinator.syntactical._

object OrderDsl extends StandardTokenParsers {
  lexical.reserved +=
    ("to", "buy", "sell", "min", "max", "for", "account", "shares", "at")
  lexical.delimiters += ("(", ")", ",")                    ❶ 词法分隔符和保留字

  lazy val order =
    items ~ account_spec                                   ❷ 顺序组合子（~）

  lazy val items =
    "(" ~> rep1sep(line_item, ",") <~ ")"                  ❸ 重复组合子，带分隔符

  lazy val line_item =
    security_spec ~ buy_sell ~ price_spec

  lazy val buy_sell =
    "to" ~> ("buy" | "sell")                               ❹ 替代组合子（|）

  lazy val security_spec =
     numericLit ~ (ident <~ "shares")

 lazy val price_spec =
   "at" ~> (min_max?) ~ numericLit

 lazy val min_max =
```

```
      "min" | "max"

  lazy val account_spec =
    "for" ~> "account" ~> stringLit
}
```

用上面的文法定义可以成功分析下面的 DSL片段：

```
(100 IBM shares to buy at max 45, 40 Sun shares to sell
➥ at min 24, 25 CISCO shares to buy at max 56)
➥ for trading account "A1234"
```

鼓鼓掌吧！刚刚我们用分析器组合子库设计了第一个DSL。稍后我们会对它进行一些再加工，让它输出反映领域抽象的语义模型。

现在我们知道了文法写出来是什么样子，也知道了它能处理什么样的语言，下面可以着手探究每一条文法规则的具体分析过程了。

正式开始之前，我们需要提一提出现于文法规则开头的那组词法分隔符（lexical delimiter）❶，这个列表里的字符在输入流中起划分词法单元的作用。另外我们还定义了一组准备用在语言里的保留字，即代码清单8-2中的`lexical.reserved`列表。本节余下的部分会对Scala库中提供的组合子予以透彻的解说，教会你用它们来搭建自己的语言。如果要查阅Scala组合子的细节信息，Scala发行包中的源代码是最完整的资料来源。

1. 每一条文法规则都是一个函数

文法规则对领域概念进行建模。规则要有合适的命名，这样才能正确传达其模型代表的领域概念。规则的主体部分采用EBNF形式记述，我们先前在ANTLR中定义上下文无关文法时，用的也是相同的EBNF形式。

每一条规则返回一个Parser，代表函数体的返回值。如果函数体是由通过组合子连接起来的多个分析器构成的，那么依次使用所有的组合子之后得到的最终结果，才是规则最后返回的Parser。目前我们假定所有的规则都返回一个Parser [Any]，等到8.3.4节我们再介绍怎样在规则体中通过定制函数返回具体类型的分析器。



我们编写文法规则，务必记住一条DSL设计的黄金守则：正确地命名文法规则，让名字反映规则建模的领域概念。在用分析器组合子设计外部DSL的过程中，文法规则起着规划蓝图的作用，是我们和领域专家一起审议的凭据。它们叙述简练、含义丰富，而且能够恰当地向领域人员传递易于理解的信息。

2. 顺序组合子

Scala语言用“~”符号表示顺序组合子。我们可以在代码清单8-2的位置❷找到它。简便起见，用符号“~”来命名，但它其实只是`Parsers[T]`类中定义的一个普通方法。

Scala允许中缀运算符的书写形式，因此`a ~ b`其实等同于`a.~(b)`。顺序组合子写成中缀形式时，语句看上去如同按EBNF形式书写的样式。另外，Scala语言的类型推断特性使得语句显得更直观。

我们再从代码清单8-2中挑选一处细节来说明顺序组合子的工作原理。在位置❷，`items`接收

到传给分析器组合体的原始输入，于是它尝试调用以items命名的规则体（或方法）来分析输入❸。如果分析成功，即产生一个ParseResult，假设叫做r1。然后序列中的下一个分析器account_spec，开始处理items余下的输入。如果也分析成功并产生ParseResult r2，那么这时“~”组合子将返回一个结果类型为(r1, r2)的Parser。

3. 替代组合子

Scala库用“|”符号表示替代组合子。替代组合子以回溯方式查找备选规则。只有当前一个分析器发生可恢复失败，并且允许进行回溯的时候，替代组合子才生效。

请看代码清单8-2的位置❹。输入首先进入第一条备选规则"to" ~> "buy"。定义在Parsers特征里面的一个隐式转换把String转换为Parsers[String]。如果分析成功，则不用理会后面的任何替代规则，分析结果直接作为buy_sell的结果返回。如果分析不成功，则尝试下一条备选规则"to" ~> "sell"。如果这次的分析成功了，就返回其结果。分析器总是按照备选规则的书写顺序决定选择的先后次序。

4. 选择性顺序组合子

这种组合子在“~>”和“<~”方法中实现，分别选择性地保留右边的结果和左边的结果。选择性顺序组合子常用于剔除那些不属于语义模型构成部分的信息。也就是说，虽然我们需要识别整个序列，但只有其中一个分析器的结果是我们感兴趣的。

再看代码清单8-2。我们以位置❸的代码来说明选择性顺序组合子的原理。~>的用法类似于~，只不过它仅保留运算符右侧分析器的结果。例中的"("在后续处理中是不需要的，因此不在结果中保留。<~方法同样与~用法相似，但仅保留运算符左侧分析器的结果。例中的")"在后续处理中也是不需要的，因此也从结果中去除。

5. 重复组合子

重复组合子用来实现重复性的结构。表8-2给出了重复组合子的各种变体。

表8-2　不同变体形式的重复组合子

变体形式	解　　释
(rep(p), p*)	重复p零次或更多次
(repsep(p, sep), p*(sep))	重复p零次或更多次，中间有分隔符
(rep1(p), p+)	重复p一次或更多次
(rep1sep(p, sep), p+(sep))	重复p一次或更多次，中间有分隔符
(repN(n, p))	重复p正好n次

代码清单8-2在位置❸处使用了重复组合子。items由一个或多个line_item组成，以“,”作为分隔符。

6. 知识点间的联系

许多开发者都有一个共同的顾虑，担心这些组合子可能难以实现。的确，我们必须先做好大量的铺垫工作，才能确保组合分析器的效果。通过抽象之间的轻松组合构造更大的抽象，始终是

我们对优秀的抽象设计的最高追求。在本章中，我们将看到，分析器组合子能够以较小的代码编写负担，建立一种具有分析器绑定能力的抽象。

这种抽象就是在第6章曾经出现的Scala的Monad。那时学到的知识现在正好派上用场，Monad化的操作将使组合子的实现大大简化。

8.2.3 用Monad组合DSL分析器

分析器组合子是一种运用函数式编程的原则来组合分析器的抽象，将简单基本的分析器组合成更大更复杂、语言识别能力更强的分析器。图8-7形象地表现了组合子的作用。

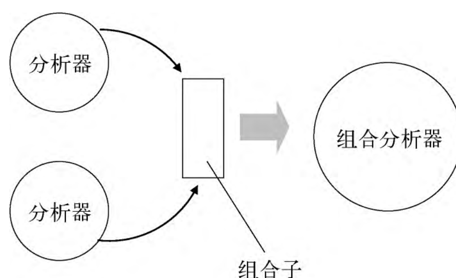


图8-7 组合子将较小的分析器组合起来，形成更大的分析器

我们从上一节得知，顺序组合子和替代组合子能把交给它们的输入串联起来。那么这种串联具体是怎么实施的呢？

1. 实现顺序组合子的笨办法

我们先来考虑Scala顺序组合子的实现方案。代码清单8-3是其中一种做法：

代码清单8-3 实现 Scala的顺序组合子

```
def ~ [U](p: => Parser[U]): Parser[~[T, U]] =
  new Parser[~[T, U]] {
    def apply(in: Input) =
      Parser.this(in) match {                                ❶ 调用当前分析器来分析输入
        case Success(r1, next1) => p(next1) match {          ❷ 成功！把余下的输入传下去
        case Success(r2, next2) => Success((r1, r2), next2)  最后成功！
        case Failure(msg, next) => Failure(msg, next)
      }
      case Failure(msg, next) => Failure(msg, next)
    }
  }
```

这个实现是正确的，组合子的行为完全符合要求。当前分析器处理原始输入流并进行分析❶。如果成功，则分析结果连同余下的输入一起被传递给参数中指定的下一个分析器进行分析❷。如果也成功了，最后的分析结果连同余下的输入一起，作为最后的分析器`~[T, U]`返回。`Parsers` trait里为`~[T, U]`定义了一个类。

那这个实现挺不错的，对吧？呃，好像哪里不对劲。你看出来问题在哪里了吗？

那些担任串场角色的代码占据了舞台的中心，这段程序的观众很难在一片喧闹的掩盖下发现表达顺序组合语义的核心逻辑。这种含义不清的错误应该时时引起我们的警觉。错误的根源出在代码清单8-3的抽象层次上，我们用了非常低层次的抽象来编程，因此导致实现细节被暴露在用户面前。

2. 用Monad消灭串场代码

观察现在的组合子实现，那些需要被遮掩起来的细节，都是一些负责连接组合各抽象的串场代码。而我们从第6章就知道，Monad特别擅长解决这类问题。Monad化的绑定操作可以帮助我们无缝地连接各抽象，它在Scala语言里的对应实现是`flatMap`组合子。因此我们可以在组合子的设计里纳入Monad的概念，依靠各种Monad化的抽象来组合分析器。为了防止低层次的具体实现细节干扰组合子的设计，Scala组合子库把`ParseResult`和`Parser`都设计成Monad化的抽象。也就是说，我们不需要自行添加任何串场用的逻辑，就可以直接把多个`Parser`和`ParseResult`抽象连接在一起。改造之后，我们的顺序组合子实现变成了一行简单的`for-comprehension`语句：

```
def ~ [U](p: => Parser[U]): Parser[~[T, U]] =
  (for(a <- this; b <- p) yield new ~(a,b)).named("~")
```

这就对了！一个看着漂亮、用着简洁的抽象就这么呈现在我们面前。



Monad跟我们的DSL实现有什么关系呢？

作为DSL设计者，我们除了使用一个库，还需要对其内部的实现技术有所了解。Scala用库的形式来提供分析器组合子，其实是在告诉我们，这些组合子本来就是准备要被扩展的。我们在现实中必定会遇到需要自行实现组合子的情况。到那个时候，通过各种Monad化的抽象设计来连接分析器的知识就会派上用场。

美好的结果总是由各种琐碎而关键的细节支撑起来的，这些知识需要你到Scala组合子库的源代码中去发掘。

编写DSL分析器的时候，可以用Monad来改善组合子的设计，这个话题我们就说到这里。接下来，话题将转到Scala组合子库的另一项特性，它也是实现复杂DSL结构必不可少的强力工具。

8.2.4　左递归DSL语法的packrat分析

到目前为止，我们研习过的若干自顶向下递归下降分析器（参阅第7章）都只能处理固定数量的前瞻符号集，这因此限制了它们所能识别的语言种类。我们的DSL很有可能含有一些超出常规自顶向下分析器能力范围的语法，这些语法或者无法被正确识别，或者识别的效率太低。LL(1)分析器（见7.3.1节）只用一个前瞻符号来搜寻适用的文法规则，而且LL(*k*)的前瞻符号集合的大小也是有限的，最多能够匹配*k*个符号。这类分析器称为*预测分析器*（predictive parser），因为它们通过预先查看输入流的内容来推测适用的规则。

还有另一类自顶向下递归下降分析器，叫做*回溯分析器*（backtracking parser）。它们通过回退并逐条尝试备选规则的方式来推断下一条适用规则。我们从上文对Scala组合子库的介绍中得知，替代组合子就有这样的能力，它会回溯并尝试我们提供的其他备选文法规则。

预测分析器速度很快，使用线性时间解析，而回溯分析器实现会轻易地退化为指数时间。请考虑下面这段用Scala分析器组合子写成的简单的文法规则，它的工作是对表达式求值：

```
lazy val exp = exp ~ ("+" ~> term) |
               exp ~ ("-" ~> term) |
               term
```

按照这个exp分析器的定义，如果第一行备选规则开头的exp成功了，但随后的输入不是"+"，那么分析器将回滚输入，改为尝试第二行备选规则。这时分析器又把第二行开头的exp重新分析一遍。在分析器找到一条完全匹配的备选之前，会反复出现重新分析的情形。这种重复性的分析过程将导致指数级的时间复杂度。

packrat分析器（见8.6节文献[3]）通过中间结果记忆（memoization）技术来解决重复计算的问题。packrat分析器会缓存它执行过的所有计算，因此当分析器再次遇到相同的计算时，就不必再算一遍，而是直接从缓存中取出结果，其时间复杂度为常数。packtrat分析器通过回溯的方式，可以处理不限数量的前瞻符号。另外，这种分析器分析算法的时间复杂度是线性的。我们将在以下几节说明packrat分析器的优势。

定义 "记忆"（memoization）是一种实现技术，通过缓存前面的计算结果来达到避免重新计算的目的。

---

1. 记忆特性提高packrat分析器的执行效率

我们应该怎样实现packrat分析器的中间结果记忆特性呢？这取决于用什么语言来实现packrat分析器。如果使用像Haskell那样默认采取缓求值（lazy-by-default）的语言，那么完全不需要为实现记忆特性做任何工作。Haskell本身就是按需求调用（call-by-need）语义来实现的，它一方面延迟求值，另一方面记忆已求值的结果，以备后续使用。Haskell通过纯函数式的、带记忆特性的回溯分析器提供最完美的实现。

Scala不是一种默认采取缓求值的语言。分析器组合子通过使用一个特殊的、带有缓存能力的Reader来显式实现记忆特性。从下面的片段可以看出，Scala的packrat分析器扩展了Parsers trait，并在其中嵌入一个实现了记忆特性的特殊Reader(PackratReader)：

```
trait PackratParsers extends Parsers {
  class PackratReader[+T](underlying: Reader[T])
    extends Reader[T]  {
    //..
  }
  //..
}
```

如果用Scala提供的PackratParsers实现来执行刚才举例的exp分析器，效率将大为提高。虽然分析器匹配第一行备选时失败了，但对该行开头的exp识别是成功的，这个识别的结果将被记忆起来。于是当分析器遇到第二行备选开头的exp时，就不必再执行一遍识别过程，而是直接从缓存中取出分析结果。由于重用了执行过的运算，packrat分析可以在线性时间内完成。

2. packrat分析器支持左递归

即使带有记忆特性，最初的packrat分析器设计照样处理不了左递归的文法规则。实际上，任何自顶向下递归下降分析器都处理不了左递归。我们可以继续用刚才的表达式求值分析器来进行说明：

```
lazy val exp = exp ~ ("+" ~> term) |
               exp ~ ("-" ~> term) |
               term
```

如果把“100–20–45”这样的表达式输入分析器会出现什么情况呢？exp分析器首先会查找记忆表，确定是否有自身的求值结果。由于这是首次尝试进行分析，记忆表是空的，表示为Nil。于是exp分析器尝试对规则体进行求值，而不幸的是规则体又以exp开头。所以exp分析器就这样陷入了无限递归的死循环。

任何左递归的规则都可以通过一个变换过程，转为等价的非左递归文法。有些packrat分析器实现可以对直接左递归的规则进行变换。但变换后的规则会较为晦涩而难以阅读，并且令AST的生成过程更为复杂。

现在的packrat分析器通过一种新的记忆技术实现了对（直接和间接）左递归的支持，该技术最先由Warth等人实现（参见8.6节文献[4]）。

Scala组合子库里的packratParsers实现了这种形式的记忆特性，可支持文法规则中的直接及间接左递归。我们会在8.3节看到用Scala分析器处理左递归文法的例子。关于该特性实现技术的详情，请在Scala源代码中查阅与packrat分析器相关的部分。除了能在线性时间内完成不限前瞻符号数量的回溯分析外，parsers分析器还有更多适合用于外部DSL实现的特性。

3. packrat分析器提供无扫描器的语法分析

典型的分析器会单独设立一个扫描器，用于输入流词法单元的划分。packrat分析器不需要单独的扫描器，其表达词法和表达上下文无关文法的形式体系是统一的。

你可能想知道无扫描器的语法分析有什么优点。首先分析器不需要安排一个单独的词法分析器抽象，因为只有一套统一的语法需要处理。由于采用packrat分析的文法在一个抽象里囊括了整个分析阶段，所以文法间的组合会较为容易。如果我们需要组合多种外部DSL，这样的设计能提高组合能力。

当然，无扫描器的语法分析也是有缺点的。为了区分保留字和标识符，我们需要为文法补充一些作为消除歧义用途的额外信息。此外，语言中的分隔符也需要额外的消除歧义处理才能被正确识别。

4. 支持语义谓词

除了packrat分析器本身具有的语法匹配能力外，我们还可以在文法规则中添加语义谓词。这些语义谓词可以根据其他语法实体的语义来判断分析是否成功。

5. 依序选择

packrat分析器不同于其他使用上下文无关文法的分析器，其替代组合子只支持依序选择。因此如果组合子的几个备选项开头部分有重合，应该把匹配长度较长的备选项排在前面。

packrat分析器用依序选择的规定来消除LR分析器下可能出现的“移进/归约”冲突和“归约/归约”冲突。

现在，我们理解了什么是分析器组合子，也知道用packrat分析器可以得到高效率的分析设计。8.4节时，我们会再回到packrat分析器这个话题，并且用它来设计一个DSL。


是否有必要把一个DSL实现内所有的分析器都设计成packrat分析器？

通常DSL只在某些局部才需要处理复杂的左递归文法规则。我们可以在这些地方使用packrat分析器，而其他地方还是用一般的递归下降分析器，Scala分析器组合子库允许我们自由地混用普通分析器和packrat分析器。

到目前为止，本章介绍了分析器组合子的基础知识，说明了怎样在函数式语言里使用这些组合子来设计外部DSL，还对一个用Scala提供的组合子库实现的交易指令处理DSL文法样例（代码清单8-2）进行了详细的分析。经过以上学习，你肯定已经意识到，用分析器组合子这样的函数式手段来设计DSL，所要求的思维方式是不同的。我们决定实现策略的前提，是掌握了各种技术的相关能力和优缺点。下一节，我们将融汇全书关于DSL设计的全部讨论内容。其中的重点是辨析内部DSL、使用分析器生成器的外部DSL、使用分析器组合子的外部DSL三者之间的差异。

8.3　用分析器组合子设计 DSL 的步骤

分析器组合子结合了EBNF文法系统的简洁特性和纯函数的强大组合能力。我们已经讲解了很多分析器组合子的特性，现在可以从一个DSL设计者的角度，试着把它们融汇起来，实际制作一套完整交易指令处理领域专用语言。

我们用不同的宿主语言如Ruby、Groovy和Scala设计过内部DSL，也尝试了分析器生成器和DSL工作台方式下的各种外部DSL设计技法。下面即将要进行的是分析器组合子的实际演练，它会成为我们放在随身工具箱里的又一件得力工具。表8-3是几种不同实现技术的对比表格，在开始设计之前，让我们先看看内部DSL设计方式和两种外部DSL设计方式之间的区别。

表8-3　DSL实现技术的对比

特　征	内部DSL	外部 DSL	
		分析器生成器	分析器组合子
完全在宿主语言内构建	是。可以完全内嵌于宿主语言（如Scala），也可以是生成式的（如Ruby和Lisp）	否。通常需要外部的分析器生成器设施（如LEX、YACC和ANTLR）	是。宿主语言必须支持高阶函数并提供分析器组合子库（如Scala和Haskell）
供最终用户使用的DSL是可直接运行的宿主语言代码	是。DSL产物是宿主语言的方法调用	否。分析设施对DSL作语法分析，然后执行每个符号所关联的函数	否。每个词法单元都被转换成一个Parser实例，然后通过组合子串联起来
最终用户需要掌握宿主语言	基本上是，异常和错误处理要借助宿主语言的设施。而且DSL本身就必须是宿主语言下的有效程序	否。DSL是通过分析器生成器产生的一种全新语言	否。DSL是借用宿主语言提供的语言处理设施建立的一种新语言

8

希望以上对比能帮助你理清概念。接下来我们就以代码清单8-2的文法设计为基础，按部就班地建立一个完整的语言模型。第一步，我们需要验证一下该文法是否真的能识别我们的语言并生成一棵分析树。

8.3.1　第一步：执行文法

观察代码清单8-2的设计，这段文法已经完整地定义了我们的交易指令处理DSL，能够完全胜任对清单后DSL脚本的分析工作。下面这段程序将依据前面的文法定义处理DSL脚本，并在分析成功时产生输出。

代码清单8-4　运行DSL处理程序

```
val str = """(100 IBM shares to buy at max 45, 40 Sun shares
  to sell at min 24, 25 CISCO shares to buy at max 56)
  for account "A1234""""

import OrderDsl._

order(new lexical.Scanner(str)) match {          ❶ 调用order分析器
  case Success(order, _) =>
    println(order)                               ❷ 分析成功
  case Failure(msg, _) => println("Failure: " + msg)
  case Error(msg, _) => println("Error: " + msg)
}
```

在这段程序里，我们调用了DSL文法中级别最高的抽象order分析器❶（见代码清单8-2的文法定义）。如果我们输入的脚本分析成功了，那么就把输出打印出来❷；否则打印出分析器产生的错误消息。单纯打印出分析器的默认输出并无太大意义，对于语言的分析也没什么实际作用。在下一节里，我们会在这个地方生成语义模型。不过现在，你能猜出打印语句❷会输出什么结果吗？

为了找到答案，我们要从分析过程产生的分析树着手。请看图8-8。

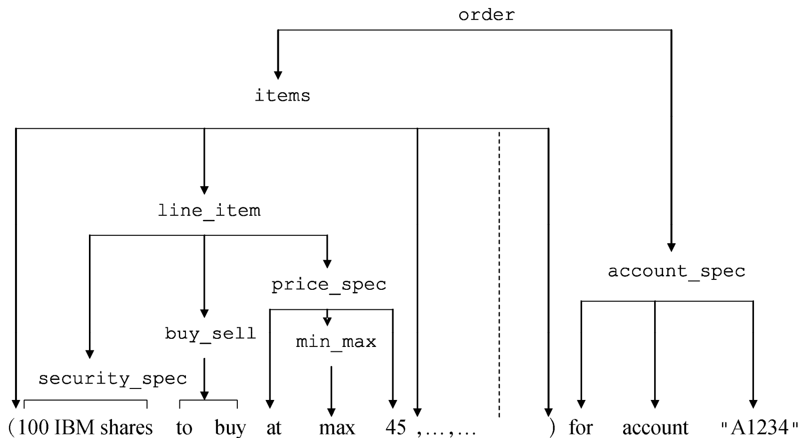


图8-8　根据代码清单8-2的文法定义产生的分析树。省略号的部分代表可以出现多个line_item成分。所有成分最终归约为树根处的order节点

这个分析过程与我们在第7章讲解用ANTLR开发外部DSL时介绍的分析过程相同。在分析过程的每一步，我们都把分析结果作为字符串输出。输出内容取决于我们搭建文法时使用的组合子。例如DSL脚本中"100 IBM shares"的部分，负责归约它的文法规则是`lazy val security_spec = numericLit ~ (ident <~ "shares")`。因为我们用了<~组合子，所以其中的"shares"部分被从输出中刈除。整个片段的分析结果由`numericLit`和`ident`的结果顺序组合而成，即输出为`(100~IBM)`。注意`rep1sep`组合子会生成一个由所有`line_item`抽象组成的`List`。可选项组合子（?）会生成一个Scala语言的`Option[]`结构。

对图8-5分析树上所有的节点都进行类似的处理之后，我们得到DSL脚本分析成功的最终输出：

```
(List((((100~IBM)~buy)~(Some(max)~45)),
➥(((40~Sun)~sell)~(Some(min)~24)),
➥(((25~CISCO)~buy)~(Some(max)~56)))~A1234)
```

这样一个纯文本的输出，真的能在现实的应用程序里起作用吗？确实不能，将一段由多元组和列表汇聚起来的DSL脚本变成无结构的文本表示，这样的分析成果没有任何现实意义。因此，我们要建立起`Order`抽象的语义模型，然后在分析过程中利用另外一些组合子向该模型填入实际的内容。

8.3.2 第二步：建立DSL的语义模型

现在我们知道，前面默认的分析结果输出毫无用处，我们需要在应用的上下文里赋予其意义和功用。那么具体应该怎么做呢？答案很简单：我们需要用一个更合适的抽象来充当DSL的语义模型。

为`Order`抽象建立语义模型并不是难题，但构建好的模型，要怎样结合到分析过程中去呢？

Scala组合子库准备了一些函数施用组合子，可以用在对分析结果的变换操作上。这些组合子帮助我们把语义模型和文法规则集成在一起。我们分析DSL脚本，同时调动这些组合子，一块一块地垒起语义模型。各分析器不再（像代码清单8-2那样）输出默认的返回值，而是返回语义模型需要的属性。这样，当分析过程完成时，作为语义模型的AST也就完整地建立起来了。

下面，我们来详细了解一下这些函数施用组合子。

1. 函数施用组合子

Scala有两个函数施用组合子：`^^`和`^^^`。跟别的组合子一样，`^^`和`^^^`都是`Parsers` trait里的方法。对于分析器p和函数`f`，表达式`p ^^ f`将产生一个识别p的结果的分析器。如果p分析成功，则组合子将对p的结果施用函数`f`。请思考下面的文法片段：

```
lazy val order: Parser[Order] = items ~ account_spec ^^
➥{ case i ~ a => Order(i, a) }
```

`^^`组合子对表达式`items ~ account_spec`的分析结果施用了一个匿名的模式匹配函数。因此`order`分析器输出的不是默认返回值，而是一个`Parser[Order]`。值得注意的是，匿名函数返回的本来是一个`Order`抽象，但由于`Parsers` trait内定义的隐式转换，被提升为相应的`Parser`类型。对于这个细节的详细说明可以参看图8-9及其后的解释。

^^^组合子类似于^^组合子，只不过^^是对分析器p的结果施用一个函数f，而^^^则是将分析器p的结果替换为一个指定的取值r。

2. 偏函数施用组合子

Scala的偏函数施用组合子是^吗?对于分析器p和偏函数f，表达式p ^? (f, error)产生一个识别p的结果的分析器。如果p分析成功，且f在p的结果上有定义，则组合子对p的结果施用函数f。如果f不适用，出现error并给出相应的理由。

我们的最终目标是解析DSL脚本并建立一个供核心应用使用的领域模型。对于交易指令处理DSL来说，Order抽象即为其中一个核心的领域构造产物。那么下一节，就让我们运用代码清单8-2定义的文法和刚刚学会的几个组合子来建立这个重要的Order抽象。

8.3.3　第三步：设计Order抽象

我们打算自底向上地构建Order抽象，这样随着语法分析的步骤进展，组成抽象的那些构造单元就正好对应地成为一个个AST节点。很容易想到，为了达到这种设想中的效果，我们可以用Scala的case类来建模抽象的构造单元，然后通过函数施用组合子直接将其插入到文法规则之中。（对Scala语言case类的详细介绍参见附录D。）

首先请看下面的Order模型。它既是语义模型，也是分析器要生成的AST。

代码清单8-5　交易指令处理DSL的语义模型

```
package trading.dsl

object AST {
  trait PriceType
  case object MIN extends PriceType
  case object MAX extends PriceType

  case class PriceSpec(pt: Option[PriceType], price: Int)

  case class SecuritySpec(qty: Int, security: String)

  trait BuySell
  case object BUY extends BuySell
  case object SELL extends BuySell

  case class LineItem(ss: SecuritySpec,
    bs: BuySell, ps: PriceSpec)

  case class Items(lis: Seq[LineItem])

  case class AccountSpec(account: String)

  case class Order(items: Items, as: AccountSpec)
}
```

这是再普通不过的Scala代码，我们在第6章设计内部DSL的时候就写过很多类似的。清单里的这些类要被分派、插入到各自对应的文法规则里，然后在实际生成AST的时候再汇合起来，组成我们现在看到的样子。

8.3.4 第四步：通过函数施用组合子生成AST

有了语义模型后，我们就可以在分析过程的每一个步骤里添加构造AST所需的Scala组合子。不管通过什么技术手段来处理DSL，最终目标总是产生一个可供应用在别处使用的抽象。

下面的代码清单在代码清单8-2的文法基础上添加了函数施用组合子。

代码清单8-6 交易指令处理DSL的AST

```
import scala.util.parsing.combinator._
import scala.util.parsing.combinator.syntactical._

object OrderDsl extends StandardTokenParsers {
  lexical.reserved +=
    ("to", "buy", "sell", "min", "max", "for", "account", "shares", "at")
  lexical.delimiters += ("(", ")", ",")

  import AST._                                          // 让分析器能够访问语义模型

  lazy val order: Parser[Order] =
    items ~ account_spec ^^ { case i ~ a => Order(i, a) }   // 函数施用组合子 ^^

  lazy val items: Parser[Items] =
    "(" ~> rep1sep(line_item, ",") <~ ")" ^^ Items

  lazy val line_item: Parser[LineItem] =
    security_spec ~ buy_sell ~ price_spec ^^
      { case s ~ b ~ p => LineItem(s, b, p) }

  lazy val buy_sell: Parser[BuySell] =
    "to" ~> "buy" ^^^ BUY |
    "to" ~> "sell" ^^^ SELL                             // 函数施用组合子 ^^^

  lazy val security_spec: Parser[SecuritySpec] =
    numericLit ~ (ident <~ "shares") ^^
      { case n ~ s => SecuritySpec(n.toInt, s) }

  lazy val price_spec: Parser[PriceSpec] =
    "at" ~> (min_max?) ~ numericLit ^?                  // ❶ 偏函数施用组合子 ^?
      ({ case m ~ p if p.toInt > 20 => PriceSpec(m, p.toInt) },
        ( m => "price needs to be > 20" ))

  lazy val min_max: Parser[PriceType] =
    "min" ^^^ MIN | "max" ^^^ MAX

  lazy val account_spec: Parser[AccountSpec] =
    "for" ~> "account" ~> stringLit ^^ AccountSpec
}
```

只要熟悉每个组合子的含义，这段代码基本上是不言自明的。在大多数规则里面，我们用^^组合子解构分析器返回的多元组，并将其嵌入到紧跟其后的匿名模式匹配函数。图8-9对一段文法规则样本的归约过程进行了剖析，从语义的角度解释了幕后发生的活动。

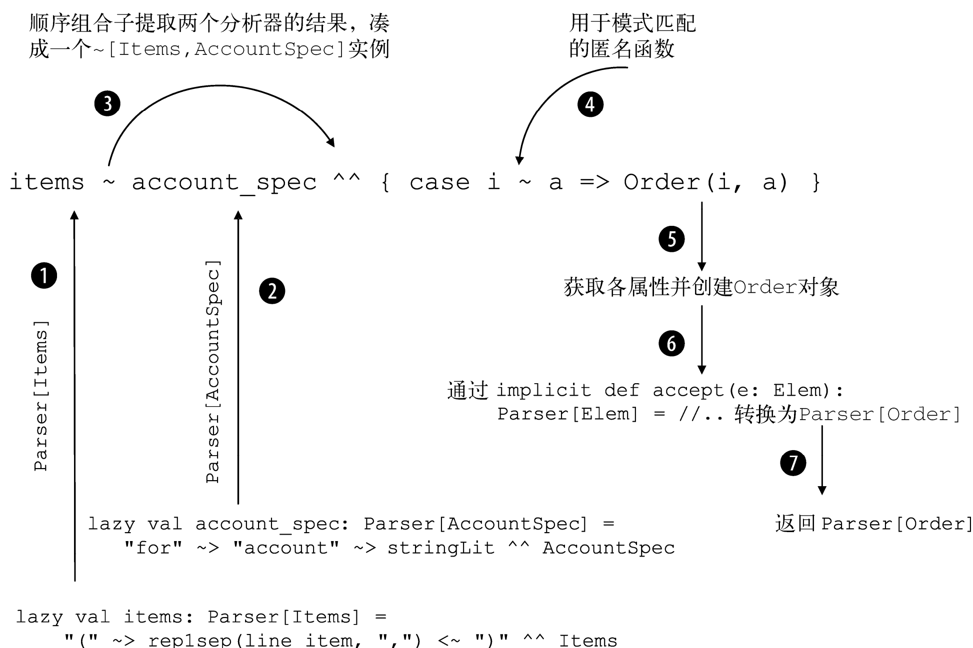


图8-9　一条规则样本从归约过程开始到最终返回`Parser[Order]`的详细步骤。`items`和`account_spec`都是上游的`Parser`，分别在❶和❷汇入此规则。顺序组合子执行`Parser[Items]`和`Parser[AccountSpec]`，并将两者的结果构造为一个~实例，传递给函数施用组合子❸。模式匹配完成后❹，一个`Order`实例即被构造出来❺。然后在隐式转换的作用下，`Order`实例被提升为`Parser`类型❻，并返回❼

图8-9有一条规则没有按照“^^组合子加模式匹配”的格式书写：

```
lazy val items: Parser[Items] =
  "(" ~> rep1sep(line_item, ",") <~ ")" ^^ Items
```

在这条规则里，我们没有通过一个匿名的模式匹配函数，而是直接使用了`Items`构造器。因为^^组合子的前一个分析器返回的是`Seq[LineItem]`类型的单一值，正好可以直接作为`Items`构造器的参数，所以我们可以采用这样的写法。`account_spec`对应的规则也采用了相同的技巧。

代码清单8-6用到偏函数组合子^了吗? ❶偏函数有可能在分析器的返回值上没有定义，针对这样的例外情况，我们预备了只在特殊上下文内生效的错误消息。请考虑这样的场景，假设脚本中设定的单价最高为10；这时分析是成功的。但是我们在偏函数的定义里，加入了验证输入的语义。比如例中的模式匹配语句内设置了验证条件，规定单价的最小值必须高于20。（附录D对Scala的模式匹配特性有进一步的介绍）那么此时在代码清单8-6的位置❶，虽然分析器报告分析成功了，但`PartialFunction`在分析器的返回值上是没有定义的。于是我们就通过这样的技巧实现了验证输入，并能针对特定情况报告错误的语义。

至此，我们的分析器已经具有完备的功能，它按照语义模型规定的结构返回的AST，同时也是一个可以直接在应用中使用的`Order`对象实例。



你有没有想过，为什么每个方法都是以`lazy val`开头，而不用`def`呢？因为对`lazy val`方法的求值会被推迟到真正使用的时刻，而且规则的定义顺序是无关紧要的。

祝贺一下自己吧！我们用分析器组合子完整地实现了一种外部DSL。这是一个简洁明了的DSL实现，其语法定义的表达形式仿照了大家熟悉的EBNF风格。其语义模型不但仅仅是由普通的Scala抽象构成的，而且完全与语法定义解耦。作为设计者，我们还能要求更多吗？

能交出这样一份答卷，证明我们已经准备好尝试更高级的练习——一种需要用到packrat分析器的DSL实现。

8.4　一个需要 packrat 分析器的 DSL 实例

上一节我们用分析器组合子开发了一个完整的DSL，期间全然没有提起过packrat分析器。Packrat分析器很特别，因为它可以做到普通的自顶向下递归下降分析器做不到的事情。我们将在这一节里开发一个需要依靠packrat分析器才能实现的新的DSL。如果说上一节的DSL让我们认识到分析器组合子函数式的威力，本节的DSL将更多地表现Scala的`PackratParsers`实现所独具的光彩。

8.4.1　待解决的领域问题

交易指令处理领域已经被我们摸索得差不多了，接下来我们打算围绕一个交易后的业务用例来建立新的DSL。

从事存管业务的金融机构可以代表客户保管证券。客户需要做的只是在金融机构那里开设一个账户，以后客户买入卖出的证券就交由该机构代为照管维护。证券及现金交易完成后，要按照一定的规则来确定保管的结算银行和账户，我们的新DSL就是给投资经理用来设定这些规则的。用这个领域的术语来说，我们DSL描述了存管机构怎样为其客户管理结算常设指示（settlement standing instruction，简称SSI）。本节随附的插入栏简要说明了这个待解决的领域问题。

 **金融中介系统：结算常设指示**

交易就意味着要在交易各方之间进行证券和货币的交换。这个交换过程发生在交易确立之后，称为交易结算。结算涉及交易各方账户之间进行资金和证券的转移。根据交易类型、证券种类、交易者的身份以及其他各种因素，结算可能涉及多个账户。

为了便于处理资产的转移，投资经理需要维护一个**常设规则**数据库，每次交易后从中查询如何进行交收的指示。这些规则就是所谓的**SSI**。SSI要经常性地公布给中介和存管人知晓。

交易结算通常由两部分组成：证券部分和现金部分。两部分的结算指示可以相同，也可以不同。如果希望证券部分和现金部分分别结算，那么相应的SSI需要明确说明这一点。

举个例子，投资银行可能会有这样的规则表述：在日本市场履行的股票交易应本行内部结算到账户A-123。这条规则将适用于在该投资银行存管的所有客户。规则可以有层级关系，查

找的时候按照从具体到宽泛的顺序。假如有另一条规则表述：对Sony的股票交易应外部结算到BOTM账户BO-234。那么综合两条规则，在日本市场履行的所有股票交易中，Sony股票的交易将通过BOTM结算，除此之外的股票交易都在本投资银行内部结算。

现实中什么人会使用这种DSL？首先投资经理是潜在的用户，其次还有投资银行内所有从事证券结算的业务人员。交易系统采用SSI有很大的好处，因为它能够将领域问题准确而精炼地向领域用户表述出来。在我们着手实现DSL之前，先来看看SSI在交易和结算流程中所处的位置。

1. 理解业务流程

为了透彻理解SSI在交易和结算流程中扮演的角色，请看图8-10和图8-11。图8-10表示交易各方之间的基本交易及结算流程。

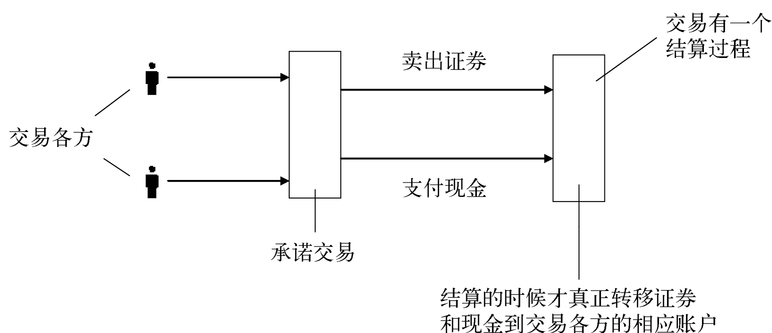


图8-10 交易和结算流程。交易是在买卖双方之间达成交换证券和现金的承诺。结算是对交易承诺的落实，相关的证券和现金被实际转移到对方的账户上

图8-11说明了为什么没有SSI信息就无法完成交易和结算流程。

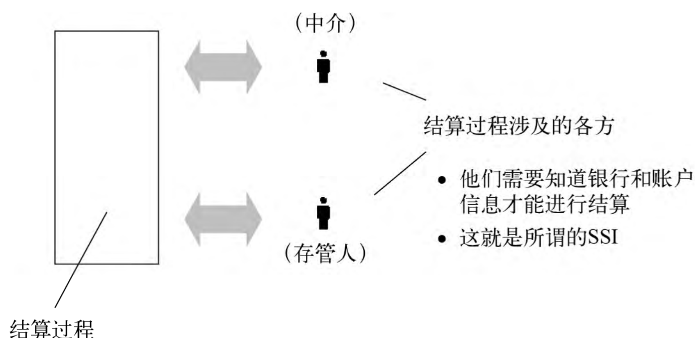


图8-11 完成结算过程需要SSI。中介及受托存管人需要掌握银行及账户信息，才知道该向何处交收证券和现金

我们对什么是SSI有了一点概念之后，可以先看几条有代表性的SSI规则样例，以对投资经理希望发布的规则有个直观的印象。

2. 将要实现的SSI规则样例

为了叙述简便起见，这里只涉及一小部分简单的规则，现实中的规则其实复杂得多。

- 客户*chase*在*JPN*市场履行的对*ibm*的交易内部结算到本行账户*a-345*。
- 客户*chase*在*JPN*市场履行的交易内部结算到本行账户*a-123*。
- 客户*nri*在*US*市场履行的交易外部结算到*CITI*账户*a-345*。
- 客户*chase*对*sony*的交易内部结算到本行账户*n-234*。
- 客户*chase*发生自账户*ch-123*的交易内部结算到本行账户*n-675*。
- 中介*icici*在*JPN*市场履行的交易，证券内部存管到本行账户*us-123*，现金外部结算到*BOJ*账户*b-954*。（这条规则对现金和证券分别指定了不同的SSI）

我们的实现依旧从文法开始。有了8.3节运用Scala分析器组合子设计DSL的经验，代码清单8-7的文法定义一点也难不倒我们。

8.4.2　定义文法

完整的文法定义会比较长，不过至少大部分是我们熟悉的写法。因此本节不再从头到尾讲解，而是聚焦到其中几个特殊之处。代码清单8-7给出了完整的文法定义。

代码清单8-7　`SSI_Dsl`的文法规则

```
package trading.dsl
import scala.util.parsing.combinator._
object SSI_Dsl extends JavaTokenParsers
               with PackratParsers {          <-- 声明使用PackratParsers

  lazy val standing_rules = (standing_rule +)

  lazy val standing_rule =
  "settle" ~> "trades" ~> trade_type_spec ~ settlement_spec

  lazy val trade_type_spec =                          <-- ❶ 左递归和依序选择
    trade_type_spec ~ ("in" ~> market <~ "market") |
    trade_type_spec ~ ("of" ~> security) |
    trade_type_spec ~ ("on" ~> "account" ~> account) |
    "for" ~> counterparty_spec

  lazy val counterparty_spec =
    "customer" ~> customer | "broker" ~> broker

 lazy val settlement_spec =
   settle_all_spec | settle_cash_security_separate_spec

 lazy val settle_all_spec = settle_mode_spec

 lazy val settle_cash_security_separate_spec =
   repN(2, settle_cash_security ~ settle_mode_spec)

 lazy val settle_cash_security =
   "safekeep" ~> "security" | "settle" ~> "cash"

 lazy val settle_mode_spec =
```

```
    settle_external_spec | settle_internal_spec

  lazy val settle_external_spec =
  "externally" ~> "at" ~> bank ~ account

  lazy val settle_internal_spec =
  "internally" ~> "with" ~> "us" ~> "at" ~> account

  lazy val market = not(keyword) ~> stringLiteral
  lazy val security = not(keyword) ~> stringLiteral
  lazy val customer = not(keyword) ~> stringLiteral
  lazy val broker = not(keyword) ~> stringLiteral
  lazy val account = not(keyword) ~> stringLiteral
  lazy val bank = not(keyword) ~> stringLiteral

  lazy val keyword =                                          将关键字建模为分析器
    "at" | "us" | "of" | "on" | "in" | "and" | "with" |
    "internally" | "externally" | "safekeep" |
    "security" | "settle" | "cash" | "trades" |
    "account" | "customer" | "broker" | "market"
}
```

从这段文法中，你能看出来为什么我们需要用到packrat分析器吗？请注意针对trade_type_spec的规则❶。没错，这个地方出现了左递归和依序选择，如我们所知，这恰好是packrat分析器擅长处理的情况。Packrat分析器因为特别采用了记忆技术（见8.2.3节），能将左递归文法的分析复杂度从指数时间降低为线性时间。

在Scala语言里实现一个packrat分析器，你需要做几件事情，请看表8-4。

表8-4　在Scala语言里把分析器变成packrat分析器的步骤

步　　骤	说　　明
(1) 混入PackratParsers	代码清单8-7的SSI分析器执行以下操作： `object SSI_Dsl extends JavaTokenParsers with PackratParsers {`
(2) 给出Reader[Elem]的具体类型，作为对分析器处理的输入类型Input的定义	Scala的packrat分析器实现依赖于一个特化的Reader实现，即Packrat- Reader。其定义为： `class PackratReader[+T](underlying: Reader[T]) extends Reader[T] {` PackratReader对内部的Reader进行包装，在其上实现记忆特性。由于我们继承了JavaTokenParsers，Reader所读入的元素类型已被定义为Char
(3) 显式指定返回类型为PackratParser[...]	不必把所有的分析器都变成packrat分析器。对于那些需要记忆特性来帮助处理回溯和左递归的分析器，显式声明其返回类型为PackratParser。我们将在实现语义模型时见到这样的例子

除了上面提到的地方，SSI_DSL的文法定义大体类似于我们曾经实现过的交易指令处理DSL。实现好的分析器还要有驱动程序才能真正运转起来，作为一个简单而实用的练习，读者可以尝试编写一个驱动程序来调用分析器并运行本节出现的一些DSL脚本。

接下来，我们讨论如何为SSI的领域抽象建立语义模型。

8.4.3　设计语义模型

我们设计的领域抽象要像8.3.4节的例子，能够直接通过函数施用组合子插入到文法规则之中。代表整个问题域的抽象被命名为SSI_AST，因为我们希望分析DSL脚本的时候就能够按照这个样子生成AST。完整的语义模型请看代码清单8-8。

代码清单8-8　SSI DSL的语义模型（即AST）

```
package trading.dsl

object SSI_AST {
type Market = String
type Security = String
type CustomerCode = String
type BrokerCode = String
type AccountNo = String
type Bank = String

trait SettlementModeRule
case class SettleInternal(accountNo: AccountNo)
  extends SettlementModeRule
case class SettleExternal(bank: Bank, accountNo: AccountNo)
  extends SettlementModeRule

trait SettleCashSecurityRule
case object SettleCash extends SettleCashSecurityRule
case object SettleSecurity extends SettleCashSecurityRule

trait SettlementRule
case class SettleCashSecuritySeparate(
  set: List[(SettleCashSecurityRule, SettlementModeRule)])
    extends SettlementRule
case class SettleAll(sm: SettlementModeRule) extends SettlementRule

  trait CounterpartyRule
  case class Customer(code: CustomerCode) extends CounterpartyRule
  case class Broker(code: BrokerCode) extends CounterpartyRule

  case class TradeTypeRule(cpt: CounterpartyRule,
    mkt: Option[Market], sec: Option[Security],
    tradingAccount: Option[AccountNo])

  case class StandingRule(ttr: TradeTypeRule,
    str: SettlementRule)

  case class StandingRules(rules: List[StandingRule])
}
```

这段Scala代码简单易懂，并不需要额外的解释。只是下一段代码通过函数施用组合子来处理分析结果时，要用到这些类，因此把它们列在这里便于参照。

在完整的文法定义里穿插处理AST的组合子，就得到代码清单8-9。这段代码最后生成的数据结构StandingRules就是我们的语义模型。

代码清单8-9　能生成语义模型的完整DSL实现

```
object SSI_Dsl extends JavaTokenParsers
               with PackratParsers {
                                                                    导入AST
  import SSI_AST._

  lazy val standing_rules: Parser[StandingRules] =
    (standing_rule +) ^^ StandingRules

  lazy val standing_rule: Parser[StandingRule] =
    "settle" ~> "trades" ~> trade_type_spec ~ settlement_spec
      ^^ { case (t ~ s) => StandingRule(t, s) }

  lazy val trade_type_spec: PackratParser[TradeTypeRule] =
    trade_type_spec ~ ("in" ~> market <~ "market")          ❶ 返回类型为
      ^^ { case (t ~ m) => t.copy(mkt = Some(m)) }  |          PackratParser
    trade_type_spec ~ ("of" ~> security)
      ^^ { case (t ~ s) => t.copy(sec = Some(s)) } |
    trade_type_spec ~ ("on" ~> "account" ~> account)
      ^^ { case (t ~ a) => t.copy(tradingAccount = Some(a)) } |
    "for" ~> counterparty_spec
      ^^ { case c => TradeTypeRule(c, None, None, None) }

  lazy val counterparty_spec: Parser[CounterpartyRule] =
    "customer" ~> customer ^^ Customer |
    "broker" ~> broker ^^ Broker

  lazy val settlement_spec =
    settle_all_spec |
    settle_cash_security_separate_spec

  lazy val settle_all_spec: Parser[SettlementRule] =
    settle_mode_spec ^^ SettleAll

  lazy val settle_cash_security_separate_spec: Parser[SettlementRule] =
    repN(2, settle_cash_security ~ settle_mode_spec) ^^ { case l: Seq[_] =>
      SettleCashSecuritySeparate(l map (e => (e._1, e._2))) }
  lazy val settle_cash_security: Parser[SettleCashSecurityRule] =
    "safekeep" ~> "security" ^^^ SettleSecurity |
    "settle" ~> "cash" ^^^ SettleCash

  lazy val settle_mode_spec: Parser[SettlementModeRule] =
    settle_external_spec |
    settle_internal_spec

  lazy val settle_external_spec: Parser[SettlementModeRule] =
    "externally" ~> "at" ~> bank ~ account
      ^^ { case b ~ a => SettleExternal(b, a) }

  lazy val settle_internal_spec: Parser[SettlementModeRule] =
    "internally" ~> "with" ~> "us" ~> "at" ~> account ^^ SettleInternal

  //..                          余下部分与代码清单8-6相同
}
```

对于出现左递归文法的`trade_type_spec`规则❶，我们设置其返回类型为`PackratParser[TradeTypeRule]`。这样它对备选项作回溯分析的时候将会用上记忆技术，左递归的问题也会按照8.2.3节的优化方式得到解决。

很有成就感吧？一个完整的DSL连同它的领域模型一起漂亮地呈现在我们面前了。文法看上去生动到位，StandingRules抽象也准确地表达了领域实体的模样。所有的分析器经过函数施用组合子的沟通串联，按照各自的层级、先后，协力建立起领域模型。

我们知道，分析器组合子的关键是函数式编程，而组合子范式的扩展性也同样源自函数之间的组合。下一节我们将看到Scala分析器对扩展DSL的贡献。

8.4.4　通过分析器的组合来扩展DSL语义

我们在8.2节讲解过，分析器可以定义成接受输入并产出分析结果的纯函数。在Scala库里，我们把这样的定义表达为(Input => ParseResult[T])。而组合子是以顺序、替代、重复等方式对分析器进行组合的高阶函数。那么，分析器和分析结果之间是怎样组合的呢？

1. 以Monad方式实现定制扩展

如果我们翻看Scala分析器组合子库的源代码，就会发现ParseResult[T]和Parser[+T]都是Monad化的结构。换言之，它们都实现了标准的map、flatMap和append方法，而这几个方法对于实现单个的组合子有很大的帮助，可以免去组合分析器时显式串接输入的麻烦。我们在对代码清单8-3的顺序组合子实现进行改造时已经体会过它们的作用，Monad化的Parser和ParseResult配合for-comprehension产生了非常精炼的顺序组合子实现。

如果能在分析器的基本抽象上附加一层组合语义，那么规则的编排将具有很强的灵活性。我们可以串联组合子，可以自定义任意的变换函数去变换分析结果，还可以在已有的分析器上添加额外的语义。举个例子，我们可能希望在交易指令处理DSL的某个分析器里记录下分析的过程。那么利用针对Parser抽象定义的log组合子，我们很容易就能做到：

```
lazy val line_item: Parser[LineItem] =
  log(security_spec ~ buy_sell ~ price_spec ^^ { case s ~ b ~ p =>
    LineItem(s, b, p) })("line_item")
```

log来自Parsers特性，它被实现为针对一个现有分析器的装饰器，可以在分析器执行前后记录信息。更详细的情况请参看Scala源代码。

2. 把分析器设计成装饰器

我们在代码清单8-6里，利用偏函数施用组合子，在分析器里添加了仅在特定上下文内生效的验证功能。但如果想在分析过程中加入更丰富的语义，我们可以把分析器设计成装饰另一个分析器的形式。请看代码清单8-10。

代码清单8-10　一个带验证功能的分析器，在分析器上加入了领域语义

```
trait ValidatingParser extends Parsers {
  def validate[T](p: => Parser[T])(
    validation: (T, Input) => ParseResult[T]): Parser[T] = Parser (
      in => p(in) match {
        case Success(x, in) => validation(x, in)
        case fail => fail
      }
    )
}
```

ValidatingParser对一个已有的分析器进行了包装，可在其上添加任意的领域语义。validate方法的参数validation是一个闭包，如果我们的DSL需要增加某些专门的领域语义的话，可以放在闭包里。稍后我们会在SSI_Dsl分析器（见代码清单8-7）上演示这种手法。

不知道你还是否记得，本章前面的插入栏里提到过，一条SSI可以分别对现金和证券的结算作出不同的指示。我们在代码清单8-9中将这种情况建模为下面的分析器：

```
lazy val settle_cash_security_separate_spec: Parser[SettlementRule] =
  repN(2, settle_cash_security ~ settle_mode_spec) ^^ { case l: Seq[_] =>
    SettleCashSecuritySeparate(l map (e => (e._1, e._2))) }
```

分析器执行后得到一个SettlementRule抽象，该case类在我们的语义模型里是这样定义的：

```
case class SettleCashSecuritySeparate(
   set: List[(SettleCashSecurityRule, SettlementModeRule)])
   extends SettlementRule
```

这条规则的分析器需要验证脚本中确实含有两段指示，这一点通过repN(2, ..)实现了。但仅仅这样还不能证明规则有效，我们还必须确定，其中一段是对证券结算的指示，另一段是对现金结算的指示。那么，应该怎么做呢？

3. 接入装饰器

其中一种办法是接入刚才实现的ValidatingParser，通过它来执行检验SSI规则有效性的领域验证逻辑。下面的代码清单对相关文法规则实施了改造。

代码清单8-11　以装饰器形式实现的ValidatingParser

```
lazy val settle_cash_security_separate_spec: Parser[SettlementRule] =
    validate(
      repN(2, settle_cash_security ~ settle_mode_spec)
➥         ^^ { case l: Seq[_] =>
          SettleCashSecuritySeparate(l map (e => (e._1, e._2))) }
    ) { case (s, in) => {
          if ((s hasSettleSecurity) && (s hasSettleCash))
            Success(s, in)                                    ❶ 验证通过
          else Failure(
            "should contain 1 entry for cash and              ❷ 验证失败
➥security side of settlement", in)
        }
    }
```

如果验证前的分析结果为Success，且得到的List内含有一个SettleSecurity和一个SettleCash对象，那么我们返回Success作为验证后的最终结果❶；否则因为没能通过领域验证而将原来的成功结果改为Failure❷。当然，为了让ValidatingParser对我们的文法规则起作用，还要将它混入到原先的SSI_Dsl分析器中：

```
object SSI_Dsl extends JavaTokenParsers
               with PackratParsers
               with ValidatingParser {
  //..
```

这种将多个分析器组合的手法是Decorator设计模式的一种应用。这样的设计可以保持基本抽象（也就是例子里的核心分析器）不受侵染，同时又能随时根据需要插入额外的领域逻辑。

8.5 小结

如果一路坚持学习分析器组合子，那么到本章结尾这里，你已经对这种函数式编程在语言设计领域的高级应用有了相当程度的了解。分析器组合子可以说是最为简洁的一种外部DSL设计手段。我们不需要为了实现DSL而自行设计一套语言基础设施。分析器组合子技术给我们提供一种内部DSL来作为设计外部DSL的手段。我们可以一边在模块化、异常处理等基本服务上借用宿主语言的基础设施，一边为用户设计全新的语言。

要点与最佳实践

- 分析器组合子在宿主语言的语法范围内提供了一种函数式色彩浓厚的外部DSL设计手段。
- 用分析器组合子设计出来的外部DSL往往拥有非常**精炼的实现**，因为中缀表示法和类型推断特性令组合子形成一种说明式的语法。
- 使用语言提供的分析器组合子库，首先要**留意库中是否提供了记忆分析器、packrat分析器**之类的特殊礼物。只有熟练掌握库的全部能力，我们才能为DSL设计效率最高的文法。

本章我们学习了在其核心语言的基础上，Scala如何以库的形式实现分析器组合子功能。普通的Scala函数组合在一起，就能让我们用极为近似EBNF的表述形式来定义文法规则。Scala库提供了非常丰富的组合子供我们处理语义模型，并从分析过程中产生定制的AST。最后，我们讨论了可以在普通自顶向下递归下降分析器无能为力时发挥作用的packrat分析器。此外我们还通过一个Scala实现示例的演练，真切体验到packrat分析器的高效率实现。

8.6 参考文献

[1] Wampler, Dean, and Alex Payne. 2009. *Programming Scala: Scalability = Functional Programming + Objects*. O’Reilly Media.

[2] Odersky, Martin, Lex Spoon, and Bill Venners. 2008. *Programming in Scala: A Comprehensive Step-By-Step Guide*. Artima.

[3] Ford, Bryan. 2002. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pp 36-47.

[4] Warth, Alessandro, James R. Douglass, and Todd Millstein. 2008. Packrat parsers can support left recursion. *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp 103-110.

[5] Newspeak. http://newspeaklanguage.org.

Part 3 第三部分

DSL 开发的未来趋势

第 9 章简短探讨我们在基于 DSL 的开发中观察到的一些未来趋势。函数式编程现在使用得越来越广泛，因为函数式抽象的组合能力相对优于 OO 抽象。我认为大量的 DSL 技术发展将在函数式编程的世界里酝酿成熟。当开发者真正认识到语法分析器组合子等技术的威力所在，一定会有更多人被吸引过来。DSL 语言工作台因为其完整的开发和维护能力，也将获得广阔的前景。本部分还将讨论 DSL 版本维护的重要话题，借鉴书中的一些实践，你可以帮助用户平稳度过 DSL 语法演变的考验。总之，揭示 DSL 世界中的种种发展趋势，正是本书第三部分即第 9 章的唯一要旨。

第9章

展望DSL设计的未来

本章内容

- 回顾本书的全部旅程
- DSL开发正获得越来越广泛的支持
- 编写DSL的工具越来越完善
- DSL还在继续向前演化

祝贺你！这里已经是本书的最后一章了。我们讨论着各种基于DSL的开发范式，不觉走过了长长的旅程。我们为了全方位地探讨DSL设计而使用了好几种语言，其中大部分是JVM语言。这些语言都是精心挑选的，兼顾了静态类型和动态类型语言，面向对象编程和函数式编程。本章我们将考察DSL开发领域一些正在被更多人认同，有可能成为主流趋势的技术。我们作为DSL开发的实践者，需要留意这些新动向，其中有一些也许会成长为实用的开发技术。

DSL的开发中有几个领域的发展吸引了DSL设计者们的关注，很值得我们讨论。图9-1是本章的路线图，我们将在旅途的最后一程中学习这些特性。

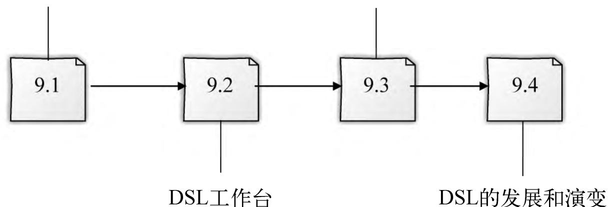


图9-1　本章路线图

本章的讨论从9.1节的语言表现力说起，这个领域的发展日新月异。Groovy、Ruby、Scala、Clojure语言的表现力都远远超越了Java，而且还在注重与使用者交流的前提下，不断地向前发展。在这些语言当中，Groovy、Ruby、Clojure等动态语言已经支持强大的元编程，甚至连有的静态类型语言也添加了元编程能力。即使我们没有马上专注于元编程，出于DSL开发人员的专业嗅觉，

也应该时时关注这方面的进展，知道各种元编程特性正在改变着当今的编程语言，正在创造一个更适合建设DSL的环境。

分析器组合子是DSL实现技术的一个进步。具备函数式编程能力的语言将会越来越多地把分析器组合子库当成语言的标准配备。分析器组合子也是9.1节的话题之一。

接着我们要说到DSL工作台，这种开发方式有可能成为今后DSL开发的常态。9.3节介绍现代IDE提供的其他辅助工具支持。最后一节我们探讨DSL的演进，学习怎样有计划地安排DSL的成长步调，谨慎维护语言的向后兼容。

如果说前面的章节谈的是DSL设计的现状，那么这一章勾画的是DSL的未来。我们要为明天做好准备，因为明天马上就会到来。

9.1 语言层面对 DSL 设计的支持越来越充分

DSL的意义在于它描述建模领域的表达能力。假如我们对一个会计系统建模，必然希望API在交流中使用借方、贷方、会计账面、分类帐、日记帐这样的专业语汇。这些术语构成了模型中的名词实体，承载着问题域的一部分核心概念。除了名词，问题域中还有动词，同样需要我们用相同的表现力水平表达出来。还记得第1章那个作为引子的咖啡店的例子吗？店员之所以能准确无误地送上我们的点单，是因为我们说的是她能理解的语言。表达的方方面面都要与建模的问题域形成共鸣。请回顾一下第3章的这段Scala代码：

```
withAccount(trade) {
  account => {
    settle(
      trade using
        account.getClient
               .getStandingRules
               .filter(_.account == account)
               .first)
    andThen journalize
  }
}
```

这里的DSL来自证券交易系统。可以看出，DSL语法很好地仿效了该领域里的*名词*和*动词*。Scala支持高阶函数，因此我们可以对领域行为（动词）和领域对象（名词）一视同仁，用相同的方式来建模。这种建模手段上的统一对语言表现力有正面的影响。

读者也许会疑惑我为什么到了最后一章，还要把“表现力”这个已经贯穿全书进行讨论的主题，又重新强调一遍。我认为有必要重提这么一个事实：对于一种能力充分的语言来说，限制其表现力的只有使用者的创造力而已。只要善用元编程、函数式控制结构等方面的惯用法，再加上一个足够灵活的类型系统，足以让程序员用领域本身的语言来描述领域问题。本节我们将讨论当前的一些语言如何拓展其表现力，从而成为DSL开发的中坚力量的。

9.1.1 对表现力的不懈追求

在这个诸多新语言争相亮相的时代，我们看到，语言给我们提供了越来越充分的支持去实现

丰富多彩的DSL语法设计。本书用了很多章的篇幅来讨论其中的Ruby、Groovy、Scala和Clojure语言，详细介绍了它们的设计能力。本节我们打算简要介绍其他一些语言的概况。这样做的主要目的不是为了探讨语言细节，而是为了让你感受现在正在发生的，众多语言为了提高自身与人交流的能力而付出的不懈努力。从图9-2可以看出一些主流语言的演变脉络，它们随着时间推演，进化成了另一种表现力更高的语言。

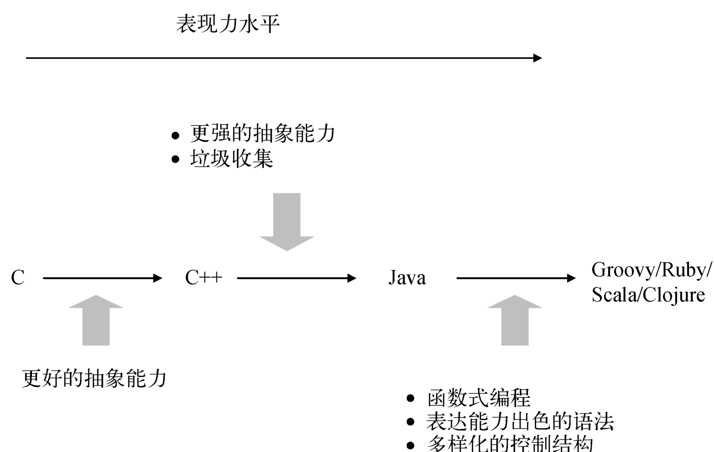


图9-2 编程语言的表现力演进过程

表现力充沛的编程语言可以帮助弥合问题域与解答域之间的鸿沟。在不支持高阶函数的OO语言里，我们只好把对象强扭成函数子（functor），才得以对领域动作建模。显然这样的间接手段会直接地反映在DSL设计结果上，造成非本质复杂性（对非本质复杂性的解释见附录A）。当函数不再是次一等的抽象手段时，DSL将去除那些由间接而产生的干扰成分，变得更干净，更容易被客户接受。

这些年里，由于编程语言表现力的提高，DSL开发实践也随之发生了变化。即使在C语言流行的年代，我们也一样做着编写领域规则的工作，只不过当时要在一个低得多的抽象层次上操作。图9-3列举了这方面的进步。

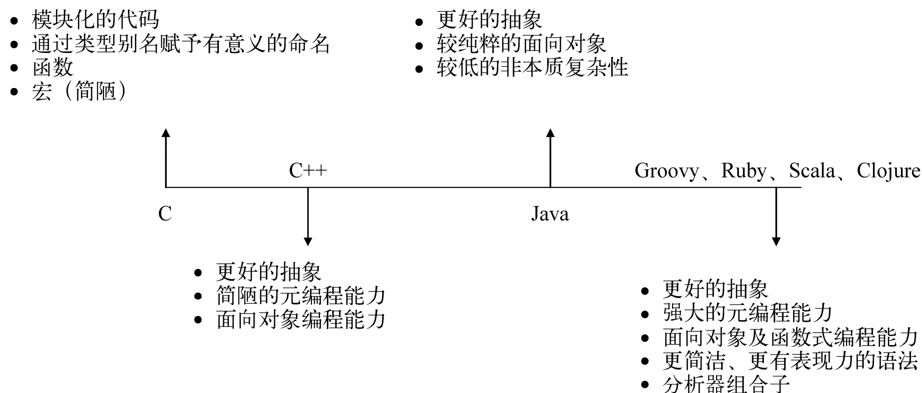


图9-3 用于DSL开发的编程语言一直在演进着，语言特性有了很大的进步

图中很多语言特性已经处于成熟阶段，但也有一些正处于争取被更多语言采纳的发展阶段。下面的三节将详述其中三种逐渐在DSL开发生态中占据一席之地的重要特性。第一种是已经在动态语言中普及的元编程。由于元编程在DSL设计方面的潜力，在众多加入元编程机制的语言当中，甚至有一些属于静态类型语言。

9.1.2　元编程的能力越来越强

观察最近发展起来的语言，我们可以发现它们的元编程能力在不断提高。Ruby和Groovy提供运行时元编程，这在第2章、第3章、第4章、第5章已经有很大篇幅的论述；JVM上的Lisp语言变种Clojure提供的编译时元编程，可以设计出表现力充沛的DSL，又完全没有运行时的性能负担。要想在DSL上有所建树，必须精通手头语言的元编程技术。

静态类型语言Haskell（见9.6节文献[10]）和Ocaml（见9.6节文献[11]）已经着手将元编程融入到语言的基础设施。Template Haskell是Haskell语言的扩展，它在语言中增加了编译时元编程设施。传统上用Haskell语言设计DSL时，一般会采用内部DSL的实现方式。开发者希望的写法和Haskell允许的写法之间往往不一致。而在编译时元编程技术下，我们可以按实际需要去设计语法，由语言设施将之转换为适当的Haskell AST结构，其机制类似于Lisp的宏。

众多语言大力发展元编程技术，直接证明DSL正在成为主流。下一节我们讲述S表达式的特性，它有潜力在大多数时候取代XML充当数据载体。

9.1.3　S表达式取代XML充当载体

有些表达形式灵活的语言，如Clojure（或Lisp），提供了S表达式（s-expression）特性，可以用代码来表示数据。现在的企业系统经常大量使用XML来描述配置数据，并冠以DSL数据建模的名头。应用中的XML结构经过适当工具的后续解析和处理，能生成可直接被程序使用的产物。这种设计除了XML难以阅读的问题之外，表达高阶结构如条件结构的能力也有欠缺，充其量只是S表达式的拙劣替代品。

我曾经做过一个项目，项目中通过XML消息来在多处部署之间传输实体。例如一个`Account`对象可以表示为下面的XML片段：

```
<account>
  <no>a-123</no>
  <name>
    <primary>John P.</primary>
    <secondary>Hughes R.</secondary>
  </name>
  <dateOfOpening>20101212</dateOfOpening>
  <status>active</status>
</account>
```

XML格式的消息要先经过解析，转换成合适的数据结构后才能供程序使用。我们不妨试试Clojure提供的S表达式，这样代码一下子就能变得清楚而简洁：

```
(def account
 {no 123,
  name {primary "John P." secondary "Hughes R."},
  date-of-opening "20101212",
  status ::active })
```

与等价的XML相比，新的片段不但语法精简，语义也更丰富，而且是一个可以执行的Clojure数据模型。我们不需要筹划额外的机制去解析它的结构，也不需要把它转换成运行时制品；这个模型直接就能在Clojure运行时中执行。我戏称这种结构是*可执行的XML*。从DSL的角度看，它不但比原来的XML版本优秀得多，而且语言定义仅仅使用了编程语言本身的特性。今后随着基于DSL的开发日益成熟，这种*数据即代码*的范式也会用得越来越普遍。

越来越普遍的还有分析器组合子，这一波趋势主要体现在函数式语言当中。我们在第8章见识过分析器组合子优秀的DSL设计能力，下面再用一节说说它的发展情况。

9.1.4　分析器组合子越来越流行

我们在第8章学过，用分析器组合子来设计外部DSL时，可以不借助任何外部工具，宿主语言的一个库就已经能满足全部要求。随着函数式编程的普及，我们将会看到分析器组合子库的爆炸性增长。Gilad Bracha开发中的新语言Newspeak（见9.6节文献[4]）拥有一个特别丰富的分析器组合子库，比我们用过的Scala分析器组合子能更好地解耦文法规则和语义模型。很多现存语言如F#（见9.6节文献[5]）、JavaScript（见9.6节文献[6]）、Scheme（见9.6节文献[7]），也都在发展自己的分析器组合子库。

分析器组合子以一种描述性的方式来定义DSL语法，外观近似于EBNF规则。虽然我们用分析器生成器也能写出类似EBNF的描述性文法规则，但分析器组合子完全在宿主语言的范围内运作，因此可以充分利用宿主语言的其他特性。宿主语言的支持将帮助我们把语义动作从文法定义中解脱出来，从而获得结构清晰的DSL实现。

除了常见的文本形式的DSL，DSL开发还存在另一个抽象层次更高的方法流派，就是DSL工作台（DSL workbench）。第7章讨论过的Xtext就是这种DSL开发范式的一个代表。将来，DSL工作台很有可能从根本上改变我们对DSL的认识。

9.2　DSL 工作台

高屋建瓴地说，DSL设计是一种在所处环境的束缚下，尽可能建立最具表现力的API的一种实践活动。对于内部DSL，我们受到宿主语言的限制。对于外部DSL，我们自行设计的语法局限于我们选用的分析器生成器或组合子。无论哪种情况，我们谈论的仍然是文本形式的DSL。不管我们提供给用户什么样的界面，它的呈现形式总是一种基于文本的结构。我们尽可能为用户改善API的表现力，但只要API是用某种语言实现的，用户终究不得不承担语言运行时强加的规则和限制。

近年出现的另一派思路不再把基于文本的DSL开发范式当做一件理所当然的事情。举个例

子，假如我是数据分析专家，那么我会希望天气预报系统的计算引擎里内嵌Excel宏。因为对我来说，电子表格才是最符合直觉的计算逻辑表达方式。而在一个单纯以文本为设计手段的DSL世界里，我们绝无可能跳出语言的桎梏，组织起像电子表格、图标引擎这样的高阶结构。

Eclipse Xtext（见第7章）等框架朝着这个方向前进了一小步。它用元模型取代纯文本格式来保存DSL，而元模型可以被投射到Eclipse编辑器。编辑器具备语法高亮、代码补全等功能。这类框架支持的抽象层次越高，就越便于最终用户自己建立、编辑和维护DSL。而协助最终用户建立、编辑和维护DSL的工具，就叫做DSL工作台。

9.2.1 DSL工作台的原理

我们曾经在第7章用Eclipse Xtext（见9.6节文献[1]）生成了一个专用的DSL工作台。JetBrains Meta-Programming System（MPS）（见9.6节文献[2]）和Intentional出品的Domain Workbench（见9.6节文献[3]）也是同类的工具。它们都放弃了单纯基于文本的编程方式，改用AST等高阶结构来作为基本的存储单元。

工作台的使用者不直接编写文本形式的程序，而是通过一种特殊形式的IDE，叫做投射式编辑器（projectional editor）来操作DSL的结构。DSL工作台通常可以和一些工具，如Microsoft Excel无缝集成，使我们能够调动这些工具去设计DSL的语法和语义。我们在Excel里建立的模型作为元数据保存在工作台的仓库里，对应着DSL里的高层次抽象。

如果需要，我们还可以从工作台仓库里保存的元模型生成指定语言的代码。这样的开发方式不正是领域用户的梦想吗？不正是我们原本赋予DSL的价值诉求吗？领域工作台也许将成为领域专家和程序开发者的理想交汇点。图9-4说明了领域工作台对DSL实现的全程支持。

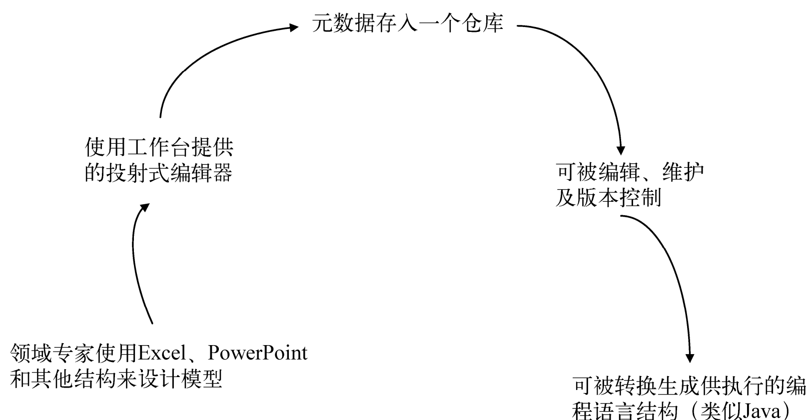


图9-4 DSL工作台为DSL实现的全部生命期提供支持。领域专家打交道的对象是像Microsoft Excel这样的高层次结构。工作台不直接保存程序文本，而是保存元数据。元数据可以被投射到一种智能的编辑器，叫做投射式编辑器上。我们通过这种编辑器来编辑元数据、处理其版本变迁以及其他方面的管理维护。工作台还拥有生成编程语言（如Java）代码的设施

在更高的抽象层次上编程，这是本书反复论述的一个主题，而且已经成为目前存在的几种DSL工作台的原则共识。它们之间的差别主要体现在抽象的表达上。表9-1总结了部分工作台产品在几个方面的差别。

表9-1　DSL工作台产品的特性差异

特　　性	工作台产品之间的差别
抽象语法的表达及定义方式	抽象语法可以用抽象语法树或者图来表达，也可以定义成元模型或文法描述形式
元模型的组合	很多工作台都支持用若干文法或元模型组合起来作为一种抽象语法的表达载体
变换能力	Xtext等工作台支持基于模板的代码变换，MPS本身直接支持从模型到模型的变换
IDE支持	大多数工作台本身即具备完善的、可定制的IDE支持，为DSL作者提供语法高亮、代码补全等上下文相关的协助功能

DSL工作台绝对是一件值得放进包里随时备用的好工具。那么我们就来说说使用DSL工作台有哪些好处。

9.2.2　使用DSL工作台的好处

不同的工作台在不同的方面各有千秋，但所有的DSL工作台都具有以下优点。

- 分离了DSL界面的关注点和DSL实现的关注点。
- 用户可与高层次结构直接互动。这些结构比一般文本型编程语言中所见的结构层次更高。因此，对于没有编程背景的领域用户来说，工作台方式下的DSL开发更有吸引力。
- 为DSL驱动的开发方式全程提供优厚的环境。
- 较容易完成多种DSL的组合。

但从发展阶段来看，DSL工作台还处在婴儿时期。虽然这项技术前景很好，也已经推广了一段时间，但开发商还需要解决好一些问题才能让DSL工作台成为主流。其中最主要的一个问题是“厂商锁定”（vendor lock-in）。DSL工作台有以下几个最为重要的方面：

- 抽象表示的呈现形式；
- 投射式编辑器；
- 代码生成器。

这几个方面全都被锁定到相应的框架上。当我们无法脱离一个特定平台来建模DSL时，顾虑在所难免。这种锁定意味着为了在项目里实现DSL，开发团队又不得不学习一套专门的工具。DSL工作台即使有这样的缺点，也仍然是一种很有吸引力的技术范式，值得我们继续关注它的未来发展。

除了寄望工作台提供完整的DSL开发环境外，我们还寻求加强IDE对DSL开发的工具支持，以改善眼下的状况。

9.3 其他方面的工具支持

如前所述，DSL工作台是首要的DSL设计工具。那么，当我们不使用工作台候，还能指望从开发环境得到什么支持呢？

我们搜寻的目光首先落在IDE上，显然这里最有可能找到适用于DSL编写的先进功能。通用语言的编程活动可以从IDE那里获得一个功能完善的编辑器，具备语法高亮、代码补全等诸多编辑功能。DSL编程也完全应该获得类似的帮助。例如，如果我们用Groovy编写金融中介系统的内部DSL，会希望IDE能高亮显示用户输入的货币代码，也会希望把自动代码补全功能应用在系统支持的金融制度上。

很多IDE虽然还不具备一套完善的DSL工作台，但已经开始提供语法高亮和自动补全这个层面的工具。现在的IDE都准备了插件架构，而且都是可扩展的。我们可以自行插入代码去实现语法高亮、代码补全等功能（见9.6节文献[8]和文献[9]）。

Contraptions for programming博客上有篇文章（见9.6节文献[8]）描述了一种在Groovy-Eclipse平台上开发DSL的插件，开发者可以自己定制实现语法高亮的功能。Groovy-Eclipse系列组件以接口形式提供一个扩展点，我们可以通过实现该接口来定制实现语法高亮的关键字。IntelliJ IDEA也包含类似可定制的Groovy DSL支持功能，其插件可实现对方法和属性的自动补全（见9.6节文献[9]）。图9-5粗略说明了如何在IDE的插件架构下为定制的DSL实现语法高亮功能。

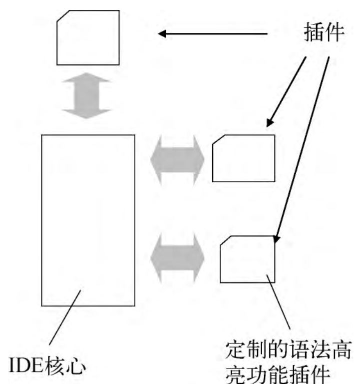


图9-5 在IDE核心之外，我们可以实现自己的插件。我们为DSL设计的语法高亮插件和别的插件一起成为IDE的一部分

一直以来，我们都围绕着DSL的开发展开讨论，而在当前的DSL环境下，DSL版本的有序演变是另一个不可忽略的重要议题。因此下一节我们将粗略探讨一下应该怎样合理规划DSL的成长轨迹，使得DSL的多个版本可以共存。

9.4　DSL 的成长和演化

我们在应用开发实践中大都使用过DSL。它的主要用途是对系统中频繁改变的组件（比如配置参数和业务规则）进行建模。而在面临改变的时候，应该按照什么样的原则去指导DSL的演化，这是一个需要我们去思考完善的问题。我们甚至在部署DSL的第一个版本之前，就应该考虑好DSL的演化策略。

9.4.1　DSL的版本化

DSL的使用情况决定了我们需要的版本化策略。如果我们的DSL只有一小群固定的、联系紧密的用户，那么不规定专门的版本化策略也是可以的。不管是修正错误还是增加需求，我们可以随时用新的版本替换旧的版本，只要随新版本附上一份标出向后兼容事项的简单说明足矣。

但如果不只一组用户在使用我们的DSL呢？那就必须规划增量式的版本化策略了。由于不是所有的用户都对新版本感兴趣，所以我们要同时采取以下两条策略：

- 代码仓库的版本管理体系必须能够同时兼顾多个发行版的维护工作；
- 必须建立专门的部署脚本，且该脚本要能够部署DSL的多个版本。

不管采取什么样的策略，至少以下问题是必须解决好的，因为任何软件模块都很容易在演进过程中遭遇这些困难：

- 处理向后兼容；
- 照顾那些不适合推广为一般情况的个别的用户需求。

我们有可能遇到的诸多困难情况实际上不一定是DSL特有的问题，而是软件部署的普遍情况。下一节我们将针对版本化过程中出现的各种难题，讨论在设计阶段可以采取的一些措施。

9.4.2　DSL平稳演化的最佳实践

假设我们在应用里使用了第三方的DSL，且应用已经部署到多个用户处。现在我们打算给应用增加一些功能，同时发现DSL的新版本刚好增加了我们需要的特性。可是，新版的DSL并不能向后兼容我们当前使用的版本。我们该怎么办呢？

再来想象一个场景。假设我们用DSL来建模证券交易的业务规则，而这些业务规则可能会因为部署地的证券交易机构而各不相同。碰巧现在东京证券交易所修改了几条规则，于是我们需要推出一个针对东京证券交易的新版本。这太可怕了！我们要同时维护几个版本，即使睡着了都要被吓醒。

还是看看有什么预防的办法吧，免得总被这些瘆人的问题闹得半夜不得安宁。

1. 隐式上下文模式更能适应版本演化

请回顾一下4.2.1节中这段基于连贯接口的Ruby内部DSL：

```
Account.create do

  number       "CL-BXT-23765"
  holders      "John Doe", "Phil McCay"
```

```
    address      "San Francisco"
    type         "client"
    email        "client@example.com"

  end.save.and_then do |a|

    Registry.registNer(a)
    Mailer.new
          .to(a.email_address)
          .cc(a.email_address)
          .subject("New Account Creation")
          .body("Client account created for #{a.no}")
          .send
  end
```

这段表现账户创建过程的DSL运用了内部DSL设计的*隐式上下文模式*。比起一个规定好所有参数顺序和数量的`create`方法，这种模式写出来的DSL更容易适应未来的演化。我们可以在不影响任何原有用户的前提下，向`Account`抽象增加新的属性。

2. 用自动代换解决向后兼容

这个策略的做法是通过设定适当的默认值，将旧的API自动代换为新的版本。例如，下面的Scala DSL片段定义了一笔固定收益型交易，这是我们在6.4.1节用过的例子：

```
val fixedIncomeTrade =
  200 discount_bonds IBM for_client NOMURA on NYSE at 72.ccy(USD)
```

用户对这段DSL很满意。交易按照脚本中指定的货币进行（即代码片段中的`USD`），这种货币称为*交易货币*。交易最后还要经过一个结算过程，我们的DSL假定结算和交易用的是同一种货币。未来某一天，规则毫无悬念地发生了变化，用户收到通知说，现在允许按照交易货币之外的另一种货币（称为*结算货币*）结算。于是DSL也相应地变成了下面的新版本：

```
val fixedIncomeTrade =
  200 discount_bonds IBM for_client NOMURA on NYSE at 72.ccy(USD) settled in JPY
```

现在的问题是，那些用旧版处理引擎写成的DSL脚本会发生什么情况？这些脚本很可能会崩溃，因为它们的语义模型里根本找不到一个表示结算货币的值。

我们可以对语义模型做一个自动代换，从而解决这个问题，把缺少的结算货币取值默认地设为与交易货币相同。这样一来，用户可以平稳地迁移到新版本的DSL，同时旧的DSL脚本也能继续顺利执行下去。

3. 门面式的DSL设计可以解决诸多版本化问题

还记得我们在5.2.1节讨论过的DSL门面吗？门面充当了模型API的保护层，方便我们调整和塑造公开给用户的语法外观。假如后续的版本需要修改DSL语法，可以将改动限制在DSL门面的范围内，这样就不会对下层的模型有任何影响。这个策略尤其适用于新版DSL仅包含少量语法变动的情况。

4. 遵从优秀抽象设计的各项原则

附录A对优秀抽象设计的原则做了很详细的解说，请你务必一读再读。只有遵循这些设计原则，用户才能和我们的DSL一起从容地面对演化。DSL的版本化和API的版本化同样重要。API越是死板僵化，就越难跟随新版本一起演变。

无论我们选择什么样的策略，都必须留有余地，让多个版本的DSL可以在同一个应用中共存。DSL开发领域还处于摸索阶段，需要更多的时间才能成熟。只要我们能多考虑DSL用户的未来需要，就是对DSL发展的积极帮助。

9.5　小结

好啦，本书的旅程到此结束。我们从所有的方面论证了DSL是一种更好的领域建模方式，又在这一章展望了基于DSL开发的未来趋势。DSL工作台以其涵盖语言完整生命期的工具集合，有望把DSL的演变之路安排得更井井有条。各种编程语言就在我们眼前一天天地提高着它们的表现力，越来越适合作为DSL的宿主语言。不管我们选择哪种语言来开发DSL，都要坚持设计原则，这样才有助于DSL增量、迭代地健康成长。

本书讨论了几种具备优秀的语言特性、特别适合用于设计DSL的JVM语言。它们除了各具特色，充分胜任DSL设计工作以外，又因为都运行在JVM之上，而获得了与Java无障碍互操作的能力。这是一个很大的优点，因为我们可以选择一种语言来满足DSL的需要，但又不会被束缚在唯一一种选定的语言上。

除JVM语言外，还有很多其他语言也被广泛地用于设计DSL。其中领跑的有纯函数式语言Haskell和面向并发编程的Erlang。软件开发圈已经认识到，只有使用那些有能力提供高阶抽象的语言，才是化解领域建模复杂性的唯一出路。而产出优美的、可重用的抽象的途径之一，正是DSL驱动的开发方式。好的DSL可以提高生产力，使代码易于维护和移植，还带给用户一个友善的界面，把所有不必要的细节都隐藏起来。总之DSL就是领域模型该有的样子。今天，我们已经在现实的软件开发中见证了DSL展露的潜力。

要点与最佳实践

- ❑ 基于DSL的应用开发相对来说还是软件行业的一个新课题。**请保持对其发展动态的关注。**
- ❑ **在基于DSL的开发领域，相关的工具支持发展得很快。**一一数来，从IDE到原生的DSL工作台，丰富的工具总是能够促进开发生态的发展。
- ❑ **所有获得关注的新语言都有一些可以用在DSL设计上的独到之处。**即使我们喜好的语言没有直接提供某些对开发和DSL实现有确凿好处的特性，我们也可以尝试去模拟它们。

9.6　参考文献

[1] *Xtext User Guide*. http://www.eclipse.org/Xtext/documentation/latest/xtext.html.

[2] *Meta Programming System*. http://www.jetbrains.com/mps/.

[3] Intentional Software. http://intentsoft.com/.

[4] Newspeak. http://newspeaklanguage.org/.

[5] Tolksdorf, Stephan. FParsec — A Parser Combinator Library for F# http://www.quanttec.com/fparsec/.

[6] Double, Chris. Javascript Parser Combinators. *Bluish Coder*. http://www.bluishcoder.co.nz/2007/10/javascript-parser-combinators.html.

[7] Pretterhofer, Lorenz. Scheme Parser Combinators. *A Lexical Mistake*. http://alexicalmistake.com/2008/06/scheme-parser-combinators/.

[8] Eisenberg, Andrew. Extending Groovy Eclipse for use with Domain-Specific Languages. *Contraptions for programming*. http://contraptionsforprogramming.blogspot.com/2009/12/ extending-groovy-eclipse-for-use-with.html.

[8] Pech, Vaclav. Custom Groovy DSL Support. *JetBrains Zone*. http://jetbrains. dzone.com/articles/custom-groovy-dsl-support.

[10] Template Haskell. *HaskellWiki*. http://www.haskell.org/haskellwiki/Template_Haskell.

[11] MetaOCaml. http://www.metaocaml.org/.

附录 A

抽象在领域建模中的角色

读者应该把这篇附录作为本书全部讨论的铺垫。DSL是覆盖在实现模型之上的一层抽象，而实现模型亦不过是在问题域模型之上，用解答域的技术平台做出的一层抽象。如果不能掌握设计抽象的正确方法，设计出来的领域模型就找不准抽象层次，那么DSL中用来表示领域模型的文法规则也会跟着出现偏差。所以，我们来看看怎么让抽象找到最适合它的位置。

A.1 设计得当的抽象应具备的特质

本节集中讨论在设计得当的抽象身上可以找到的一些特质。讨论中会用软件工程和程序设计领域作为参照来帮助理解，不过重点是如何拥有一份精心设计的抽象，协助你更轻松地构建出可重复利用的领域模型。随着讨论逐渐深入，你将理解并学会欣赏，抽象在设计一个复杂的领域模型中所扮演的中心角色。经过一系列提取抽象的练习，你将能够越来越熟练地从嘈杂的细节中提炼出模型的核心概念。为此，我们首先要讨论几种特质，正是它们把优秀的抽象设计和失败的抽象设计区别开来。



本节用易于理解的语言，讨论设计得当的抽象所应具备的若干优良品性。我会一边讨论，一边列出代码片段来演示这些优良品性在具体实现中的不同侧面。讨论到某个侧面的时候，我会选取最能充分展现其特征的编程语言来作解释。虽然面向对象编程范式占据的篇幅最大，但也有不少例子用了函数式编程原则来实现。如果你对某些例子所用的语言不熟悉，也不用急着去翻书架。因为都是一些简单而且符合直觉的例子，不需要对具体语言有深入研究，也能顺利地理解其中的设计原则。万一需要了解某些语言特性，附录C到附录F已经为你准备好了。

每一个抽象都向其客户提供一个功能。为了完成其功能，抽象向外公开一套契约（也叫做接口）供客户使用。契约根据客户的性质可以有各种变化。契约背后都有对应的实现，而且通常以抽象化手段把实现对客户隐藏起来，客户只能看到公开的契约。

接下来的三节将分别对设计得当的抽象所具备的几种特质作初步的介绍，稍后还会进行深入的讨论。

A.1.1　极简

根据客户的性质，你可能决定暴露一定程度的实现细节。但问题是，暴露出来的细节一旦公开给客户，就会和客户耦合在一起。所以要确保暴露的细节是为了实现对客户承诺契约要求的必不可少的要件。A.2节讨论到抽象的极简特质时，会再详谈这个话题。

A.1.2　精炼

设计得当的抽象，必定不能含有核心关注点以外的任何非必要细节。抽象的实现应该达到足够的纯粹度，尽可能减少细节，但又不损失必要的意义。使抽象具备这种性质的过程，是一个精炼过程，我们将在A.3节详谈。

A.1.3　扩展性和组合性

所谓工程，讲求用模块化的方式来设计事物，并能通过组合进行扩展。除了外部的组合，软件抽象还要能够从内部进行扩展。对于今天设计的抽象，未来也许需要对其补充额外的功能。重点是补充进来的部分不能影响到已有的用户。A.4节将详细探讨如何用时下的编程语言，实现能够无缝扩展的抽象。

扩展性只有通过组合性来达到。行为得体的抽象，可以组合起来构成更高层次的抽象。但是，怎样才能设计出方便组合的抽象呢？如果抽象的副作用波及到整个执行环境，会发生什么事情？

当你清楚地知道好的抽象具有哪些特质时，对于抽象在领域模型设计中所起的作用就有了评判能力。理解了这些特质，你就会认识到，为什么要把抽象摆到正确的层次上才能保证模型和领域有共同语言。只有这样，你写出来的代码，其表现力才不亚于领域专家所说的行话。

A.2　极简，只公开对外承诺的

假设你要在金融中介系统中设计一个抽象，它的功能是根据设定的一组价格类型，对外公布某种金融票据的不同类型的价格。市场中交易的每一种票据都有好几种价格，比如开盘价、收盘价、当前市价，等等。抽象应该有一个`publish`方法，可以给它指定一种票据和一个价格类型的列表作为参数。该方法返回一个`Map`，其中的键是价格类型，而值是给定票据的相应价格。我们首先会写出这样的Java程序：

```
class InstrumentPricePublisher {
    public HashMap<PRICE_TYPE, BigDecimal> publish(Instrument ins,
        List<PRICE_TYPE> types) {
            HashMap<PRICE_TYPE, BigDecimal> m =
                new HashMap<PRICE_TYPE, BigDecimal>();
        //..                                          ❶ 填充HashMap
        return m;
    }
}
```

你的本意是让publish方法返回一个Map，内含给定票据的价格类型和对应价格。但上面的实现返回了HashMap，它是方法内部用来存储数据的一个特化的Map抽象。这个特化的抽象把内部的实现暴露给了客户。返回HashMap❶成了公开的契约，那么客户代码就和HashMap耦合起来了。

假如后来需要把内部的数据结构换成TreeMap，该怎么办呢？没办法，那样做会破坏已有的客户代码。所以抽象就失去了演化的能力。怎么避免这样的问题呢？

A.2.1　用泛化来保留演化余地

马后炮总是很响，眼前就是这么个情况。当初的抽象设计应该在满足契约承诺的前提下，返回最宽泛的类型。契约原本承诺的是返回一个Map，一种支持键值对和查找策略的数据类型。所以，下面才是正确的写法，尽量不暴露内部实现，并且把返回类型调整到恰当的层次：

```
class InstrumentPricePublisher {
    public Map<PRICE_TYPE, BigDecimal> publish(Instrument ins,
        List<PRICE_TYPE> types) {
        Map<PRICE_TYPE, BigDecimal> m = Collections.emptyMap();
        for(PRICE_TYPE type : types) {
            m.put(type, getPrice(sec, type));
        }
        return m;
    }
}
```

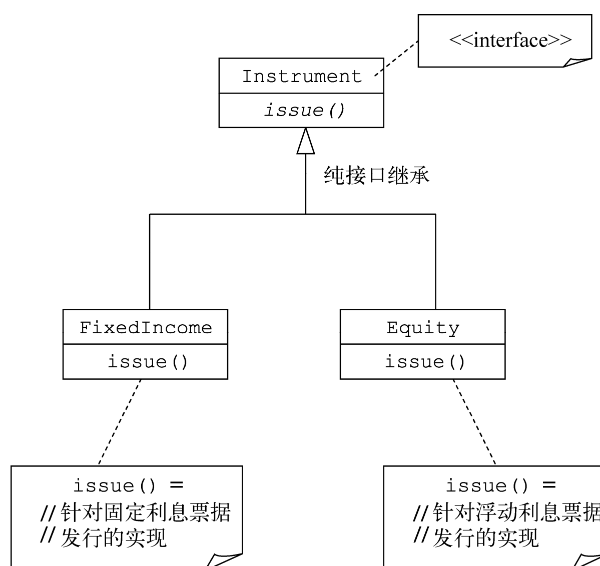
看过具体的例子后，你觉得哪些关键症状可以让你察觉抽象违反了最少公开原则？下面，我们来探讨一些有助于诊断的基本概念。

A.2.2　用子类型化防止实现的泄露

我们都知道，面向对象方法用继承手段来对抽象的共性和变异性建模。在基底抽象中定义的行为，可以被下级抽象用覆盖的方式进行细化。在由此产生的层级结构中，越靠近叶子部分，抽象就越精细，公开的契约也更为特化。用继承的概念来对子类型化建模，正好表现出抽象逐级细化的情形。子类型化，顾名思义，子类型的契约必须从父类型那里继承得来，但仅限于继承契约，契约的具体实现是子类型自己的事情，图A-1演示这个概念。

较为特化的类型称为一般化类型的子类型。子类型化并不表示上下级的类型会共用同一个实现。基于“类”（class）的面向对象语言，如Java和C#通过接口继承（文献[2]）来完成子类型化。可是在大多数基于类的面向对象语言中，“子类型化”（subtyping）往往被当做“子类化”（subclassing）的同义词，并因此导致混乱的语义和脆弱的抽象层级结构。如果使用得当，接口继承是一件有力的工具，可以在有亲缘关系的抽象族内建立牢靠的类型层次结构。如果单纯通过子类型化手段扩展抽象，绝无泄露实现之虞，同时得到的抽象是极简特质的最佳范本。

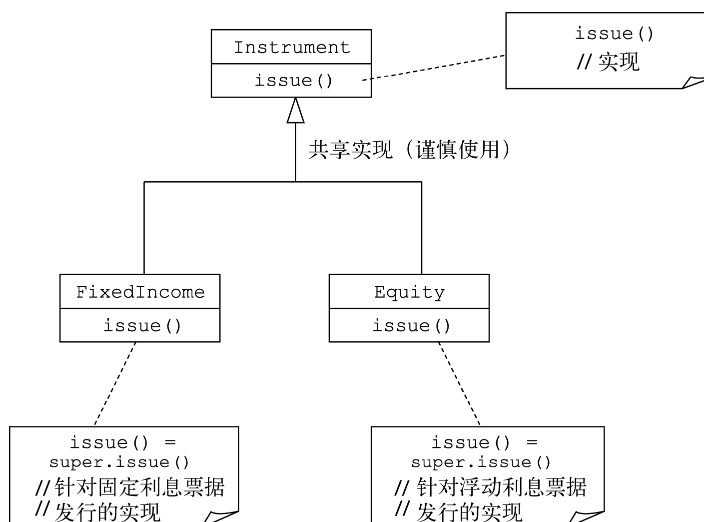
通过继承来对抽象的行为进行细化，很难避免在上下级抽象之间共享实现的情况，这种情况一般称为实现继承。



图A-1 通过接口继承完成子类型化。子类型`FixedIncome`和`Equity`仅从父类型`Instrument`继承其接口，然后自行提供实现

A.2.3 正确实施实现继承

如果能正确实施，实现继承是一种非常有用的技巧。但这种技巧很容易被滥用，一不小心，子类就随意地与基类实现产生耦合。图A-2演示了这种状况。



图A-2 实现继承产生的耦合。`FixedIncome`和`Equity`的`issue()`方法重用了基类的实现

图中的做法使子类也成了基类的客户，而且基类的实现被泄露到子类。这就产生了OO建模中称为脆弱基类的问题，对基类实现的任何修改，都有可能打破子类的契约，因而使抽象的演化成为近乎不可能的任务。这个例子的情况和前面的InstrumentPricePublisher相似，根本问题就是基础抽象把实现公开给客户。

从本节的例子可以总结出一个重要原则：*只公开有必要的部分，并且只向有必要的对象公开*。不遵从此建议，就有曝光过度、实现泄露的危险，也违背极简原则。

A.3　精炼，只保留自身需要的

在A.1节的讨论中，我们说过，抽象应该只向其客户公开核心的、必不可少的部分，这样从外部看起来，抽象是简洁的；而抽象内部的设计是否简洁，也同样重要。所谓精炼，即指从事物中萃取其本质的过程。对抽象设计来说，精炼是指去除实现中的*非本质细节*，使抽象的实现保持纯粹的过程。

A.3.1　什么是非本质的

你肯定会问，如何才能知道哪部分实现是非本质的？我引用专家的话来回答：带着深刻的认识和清醒的头脑去研究你的抽象，有过这样的经验，自然能够辨别出来。还可以下一个非正式的定义：若抽象中的一处细节不能映射到某个核心事项，那么该处细节就是非本质的。

假设你打算找一份领域建模师的工作，于是打开文字处理软件起草求职申请。你打字的同时，软件通过内置的拼写检查器标出不正确的字词。那么，你什么时候关心过软件自带拼写检查器的确切版本？又有哪次拼写检查器没有随软件启动，而需要你专门开启？理所当然地，你会假定软件打开的时候，拼写检查功能已经准备就绪，将会正确无误地运行。如果每打一个字都要特地检查、开启一遍，反而说明流程中含有非本质的细节。

如何提炼抽象，去除非本质细节？对于这方面基本概念的解释，会再次用到基于类的面向对象编程中的一种常见模式。例子照旧来自金融中介系统领域，如果对此领域不熟悉，可以翻阅第1.2和1.3节的插叙，那里介绍了一些相关的概念。

A.3.2　非本质复杂性

交给你一个任务：设计一个TradeProcessor抽象，当它收到提交的一组交易时，将计算出各种交易细节，包括交易净值、应付的各种税费和佣金，还有与交易市场相关的其他信息。你设计出来的抽象大概如同代码清单A-1的样子。

代码清单A-1　TradeProcessor处理各种交易细节

```
class TradeProcessor {
    private SettlementDateCalculator calculator;
    public TradeProcessor() {
        try {
```

```
            calculator = new SettlementDateCalculatorImpl(..);
        } catch (InitializationException ex) { //.. }
    }
    public void process(List<Trade> trades) {
        for(Trade trade: trades) {
            calculator.settleOn(trade.getTradeDate());
        }
        // 其余处理过程
    }
}
```

TradeProcessor需要按照“交易充实”过程的规定计算结算日，该日期的计算牵涉成交前后的各种因素，由SettlementDateCalculator负责提供计算服务。假设SettlementDate-Calculator是一个独立的接口，SettlementDateCalculatorImpl是其实现，负责根据成交日期及其他上下文信息确定结算日。TradeProcessor的构造器创建一个SettlementDate-CalculatorImpl的实例并保存在上下文中，供process方法后续使用。也就是说TradeProcessor实例初始化了一个服务，就要对其全生命周期负责。Settlement-DateCalculator Impl的构造器可能很复杂，需要其他服务的协助才能成功初始化。

万一哪个服务初始化失败了呢？如果遇到这样的情况，TradeProcessor类要负责在其构造器中处理因此发生的异常连锁反应，并安排相应的恢复措施。那么，TradeProcessor作为一个领域抽象，应不应该由它来考虑这些事情呢？

TradeProcessor是一个领域对象。那么在设计它的时候，就应该只把它当做一个领域对象来规划，考虑这个领域对象如何配合其他领域对象和领域服务，完成它向客户承诺的职责。至于如何实例化，如何管理各种服务的生命周期，这些都不是领域抽象的核心职责。从上面的例子可以看出，领域对象的设计很容易把一些非本质的复杂性也参杂在内，而其实把这些方面放到更低的架构层次去处理更加合适。Fred Brooks把这种情况称为*偶然复杂性*（accidental complexity，参见A.6节文献[1]），也有人称为*次要复杂性*（incidental complexity）。



只有当抽象中的非本质复杂性被限制到最少时，才能保证抽象处于一个恰当的层次。程序设计者应该注意把实例化和相关服务的生命周期管理委托给DI（依赖注入）容器等底层框架。

对设计过程的每一步都应该进行回顾，检查一下抽象是否足够精炼，是否有低层的细节被泄露到高层的抽象。如果发现像TradeProcessor类一样的情况，就知道应该重新调整上下层架构的职责分配，消除非本质复杂性。

A.3.3　撇除杂质

消灭非本质复杂性的药方，还是那个解决了很多计算机科学问题的老法子——在实现语言和领域抽像之间引入一层新的间接层。新的间接层将领域抽象隔离起来，保护它免受非本质复杂性的侵染。

在思考撇除杂质的方法之前，我们先换个角度，看看哪些成分属于杂质。我们从抽象的代码中截取另一段来进行讨论，应该撇除的信息已经做了标注。

代码清单A-2　标注了非本质细节的TradeProcessor 版本

```
class TradeProcessor {
    private final SettlementDateCalculator calculator;
    public TradeProcessor() {
        try {
            calculator = new SettlementDateCalculatorImpl(..);
        } catch (InitializationException ex) { //..  }
    }
}
```

❶ 管理生命周期的代码

❷ 服务失败时的异常处理

TradeProcessor在构造器中实例化SettlementDateCalculator的一个具体实现❶，担负了管理SettlementDateCalculator服务生命周期的职责。因此产生以下后果。

- 从此TradeProcessor对该服务的某个特定具体实现产生依赖。

  为TradeProcessor编写单元测试时，必须保证该服务实例已经就位，还要保证所有被依赖的服务全部就位。这违反了抽象应该可进行独立单元测试的原则，而且一旦脱离这个具体的实现环境，抽象被重用的可能性大大降低。

- TradeProcessor的构造器被SettlementDateCalculatorImpl的初始化逻辑引发的错误处理代码污染❷。

  代码中充斥着噪杂的细节，这些细节不应该成为TradeProcessor的首要关注方面。

对于哪些成分属于抽象中的非本质细节，你有所了解了吗？很好！让我们来消灭它们吧。

A.3.4　用DI隐藏实现细节

我们要做的就是从TradeProcessor的代码中去除涉及SettlementDateCalculator生命周期管理的部分。正确的方式是，当TradeProcessor需要SettlementDateCalculator时，我们从外部给它提供一个实例。TradeProcessor不需要关心收到的实例属于哪个确切的具体实现，因为相关服务的实例化、管理和终结等具体事务将与TradeProcessor隔离。DI（dependency injection，依赖注入）会替我们完成隔离工作。

定义　DI框架是一种外部容器，它通过说明式的配置代码，创建、装配、连接各种依赖项，把它们组织成一个对象图。关于各种DI技术的详细说明，请参阅A.6节文献[2]。

我们使用Google提供的Guice依赖注入框架（http://code.google.com/p/google-guice/），把依赖绑定到SettlementDateCalculator的具体实现。下面的代码是抽象精炼之后的样子。

代码清单A-3　精炼后的TradeProcessor

```
class TradeProcessor {
    private final SettlementDateCalculator calculator;
    @Inject
```

指示注入位置的标注

```
    public TradeProcessor(SettlementDateCalculator calculator) {
        this.calculator = calculator;        <-- 干净的构造器
    }
    //同代码清单A-2
}
```

现在，TradeProcessor摆脱了非本质的细节，实例化逻辑也被移交到外部框架。目前来说，你只需要了解如何绑定TradeProcessor类和SettlementDateCalculator的具体实现即可。我们需要在Guice里面配置一个外部Module，Guice会在应用程序启动时完成绑定。Guice的工作原理在此不作介绍，重点是通过引入一个外部框架，得以去除原先抽象中的所有非本质细节。

精炼抽象、撇除非本质复杂性的手段有很多，DI仅仅是其中一种。很多语言实现直接提供了强有力的语言特性，不必借助外部框架来达到这样的目标。第6章讲述Scala语言强大且可扩展的静态类型系统时，就有这方面的例子。函数式编程里的高阶函数和闭包可以把功能外部化，也就是把非本质细节放到抽象外部去处理。第5章详细讨论了如何使用函数式编程特性来设计领域模型，也将深入介绍了这方面的例子。

A.4　扩展性提供成长的空间

按照A.2节和A.3节的原则实施，我们的抽象已经具备了合适的曝露度（极简）和纯粹度（精炼）。可是现在客户又要求给程序添加新功能，所以你不得不在抽象中加入额外的行为。那么现在是时候检验一下抽象的扩展能力是否足够。

A.4.1　什么是扩展性

扩展性的作用是让抽象能以逐步逐块的方式成长，同时不影响已有的客户。不同的开发范式具有不同的扩展性机制。本节将总括性地介绍如何运用面向对象编程和函数式编程领域较为流行的技术来设计具备扩展能力的抽象。扩展性的其他表现形式将在介绍高级语言特性和高层次抽象时讨论，详情请参阅第5章至第8章。

Java中的Map抽象提供了一种关联性数据结构的基本功能，并向其客户提供一个Dictionary接口。标准JDK类库对java.util.Map进行了好几种扩展。其中一些是具体的子类实现，例如HashMap和TreeMap提供了Map接口的全部操作，但内部采用不同的底层存储机制。另一些是对Map的子类型化，例如SortedMap和ConcurrentMap提供了比Map类型更丰富的行为语义。

那么java.util.Map的扩展能力如何呢？假如要给java.util.Map增加一项特定的行为，并且要求对Map接口的所有变体和实现起作用，应该怎么做呢？请仔细考虑一下这个问题，因为就Java语言提供的扩展手段而言，并不容易找到答案。

仅对HashMap之类的某个具体实现进行扩展并不可行，因为那样的话，Map接口的其他实现还是得不到新功能。

还可以用一个装饰器把Map的实例包装起来（参见A.6节文献[3]），但这种方案只有在特定的

使用场景下才有效。在其他用例中，Map被包装之后，会失去原始Map实例的一些重要内容，例如SortedMap和ConcurrentMap。考虑以下的例子：

```
class DecoratedMap<K, V> implements Map<K, V> {
    private final Map<K, V> m;
    public DecoratedMap(Map<K, V> m) {
        this.m = m;
    }
    //..
}
```

❶ 包装Map

实现Map<K, V>

DecoratedMap<K, V>装饰被包装对象Map<K, V> ❶，此时即使客户在构造器中传入ConcurrentMap或SortedMap类型的实例，包装器也无法使用子类型提供的额外功能。Eugene在A.6节文献[5]讨论了一种应用场景，其中为全部Map实现提供扩展功能的唯一办法，是编写一个独立的工具函数，虽然按照纯粹主义者的观点，这是完全不符合面向对象的方式。

最后的对策唯有从头实现Map接口，但这样做将产生大量复制黏贴的重复代码。

看过所有的可能性之后，你可能会疑问用纯正的OO方式扩展java.util.Map为什么如此困难。

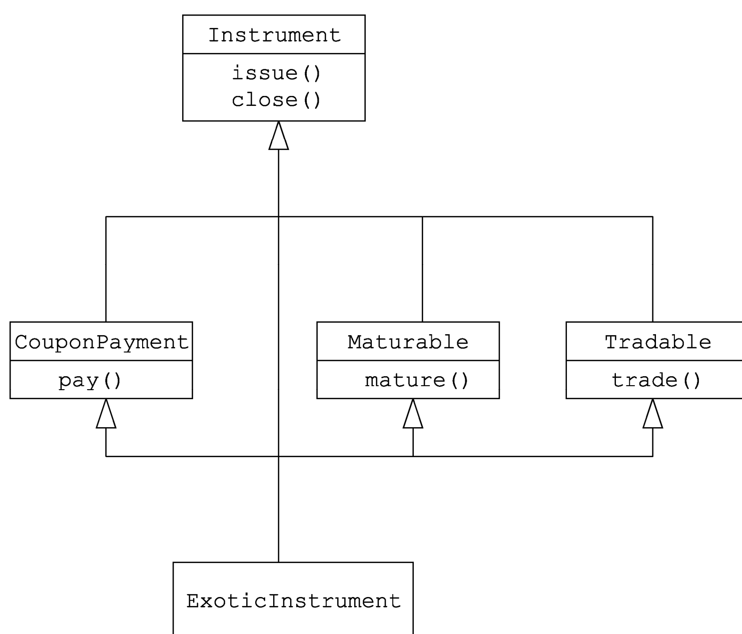
我们面对由抽象组成的一个层级结构，试图在java.util.Map类型的所有实现中插入一个新的行为。解决这个问题迫切需要实现继承的帮助，更确切地说，需要*多重实现继承*的帮助，因为这其实是一个行为的相重性问题。我们需要一种组合*独立的*小粒度抽象手段，通过把它们无缝地附加、合并到主抽象之上，来实现引入新行为或覆盖已有行为的目的。mixin提供了一种可行的途径，请看下一节。

A.4.2　mixin：满足扩展性的一种设计模式

mixin正是我们所需要的手段。有不少语言提供了这种特性，利用mixin便于混合、搭配的性质，帮助开发者建造更大规模的抽象。

假设提供服务的金融中介公司决定在市场中引入一种新的票据，名叫*热带风情票据*。这种票据的一系列特性其实在一般的票据中也很常见，因此都已经在领域模型中实现过了。剩下的工作是把各种已有的特性组装起来，为新票据扩展抽象。用基于mixin的编程方式来实现的话，简直就像mixin的字面意义一样，把各种单独的特性“混入”基本抽象，就能组合成我们的热带风情票据。从图A-3可以很清楚地看出它们是怎样“混入”的。mixin类CouponPayment、Maturable和Tradable混入父抽象Instrument，为完整的类ExoticInstrument提供行为和实现。

Gilad Bracha是一位计算理论学家，在他提交到1990年度OOPSLA（面向对象编程、系统、语言及应用）大会的论文（参见A.6节文献[4]）中，mixin被定义为*一种抽象子类，可对一系列多样化的父类进行行为特化*。mixin不可以被单独使用，所以把它们叫做抽象子类。mixin定义统一的类扩展，然后无缝地附在一个抽象家族身上，为整个抽象家族增加同样的行为。Scala（http://www.scala-lang.org）通过traits的形式实现mixin，而在Ruby中则叫做Modules。trait可以理解为Java中的接口，只不过接口中的方法声明还可以包含可选的实现给基类共享。



图A-3　基于mixin的继承。ExoticInstrument从Instrument获得issue()和close()的实现，然后与CouponPayment、Maturable、Tradable等mixin组合

A.4.3　用mixin扩展Map

现在来看看怎样用mixin给Map抽象的所有实现增加一项特定行为。假设需要给各种Map添加一个同步的get方法，覆盖原本的API。首先，在Scala中定义一个trait，在标准Map接口的基础上提供一个新增行为的实现。

```
trait SynchronizedGet[A,B]  extends Map[A, B] {
  abstract override def get(key: A): Option[B] = synchronized {
    super.get(key)
  }
}
```

然后这个行为可以混入Scala的任意一种Map类型变体，无论其底层如何实现：

```
val names = new HashMap[String, List[String]]
              with SynchronizedGet[String, List[String]]     <- 对Scala HashMap 进行混入

val stuff = new scala.collection.jcl.LinkedHashMap
➥[String, List[String]]
              with SynchronizedGet[String, List[String]]     <- 对Java LinkedHashMap 进行混入
```

注意，在最后的实现中，SynchronizedGet这个trait是在运行时对象创建期间被动态混入的。

Scala trait还可以作为多个属性的组合体，静态地混入已有的抽象。Scala的trait混入手法既能

达到实现继承的目的，又避免了Java实现的缺点。第6章介绍Scala语言特性的时候会更进一步讨论Scala trait。

A.4.4　函数式的扩展性

很多人抱怨OO编程迫使他们编写很多不必要的类。“一切都是类”并不成立，虽然面向对象有时候希望你这样想。在现实世界中，很多问题更适合建模成函数式的抽象或者基于规则的抽象。

假设要建模一个有限步骤的算法，如下面的代码片段。我故意省略了作为算法输入的参数。

```
def process(...) = {
  try {
    if (init) proc
  } finally { end }
}
```

init是算法的初始化部分。如果init成功完成，接着调用负责核心处理的部分proc。end是终结部分，负责清理各种资源。

现在要求你进行扩展，让init、proc和end都有不同的实现。一种思路是为每个步骤分别定义一个对象，把各阶段的流程包装成functor或者函数对象。这个思路是可行的，但如果能够发挥函数式编程和高阶函数的作用，会得到更好的结果。你可以把init、proc和end建模为函数，然后作为闭包传递给主流程。如下所示：

```
def process(init: =>Boolean, proc: =>Unit, end: =>Unit) = {    <- 通用的算法
  try {
    if (init) proc
  } finally {
    end
  }
}
def doInit = { //..  }          <- 初始化
def doProcess = { //..  }       <- 核心处理
def doEnd = { //..  }           <- 终结
```

最后我们来看一种非常规的扩展方式。并非所有语言都支持这种手法，但如果能理智地使用，它将是一件高效的工具。

A.4.5　扩展性也可以临时抱佛脚

扩展不一定需要建立新的抽象。不少语言提供一种叫做开放类（open class）的特性，你可以向这种类注入新的方法或者修改类中已存在的方法，直接扩展其现有结构。Ruby和Groovy都支持开放类，允许你打开任何一个类去修改其行为。一般把这种做法叫做猴子补丁（monkey patching）。很多开发者认为猴子补丁极不安全，对它大皱眉头。如果能负责任地运用，这种技术可以发挥极大的威力，可是当前的软件开发中能找到许多实际例子，让它的自我标榜站不住脚。

Ruby猴子补丁的主要问题在于缺少语法作用域;加在一个抽象上的任何东西都会进入全局命名空间，并对该抽象的所有用户可见。Scala在这一点上要好很多，它提供一种限制词法作用域的开放类，在Scala的术语里面叫做implicits，通过在语法作用域内的隐式转换来扩展现有类。（对这种语言特性的讨论可参阅http://debasishg.blogspot.com/2008/02/why-i-like-scalas-lexically-scoped-open.html）这种特性出乎意料地有用，能够在不安全的Ruby猴子补丁和严格封闭的Java类两个极端中间找到完美的平衡点。

A.5　组合性，源自纯粹

有大量的研究工作尝试将简化形式的人类语言教授给其他灵长类动物。大猩猩和黑猩猩能学会由符号和手势组成的语言，并用来沟通，当然这样的语言只是人类原始语言的一种简化组合。虽然它们可以学会越来越多的词汇，但始终没能发展出把语言组织成句子的能力。人类大脑有一个部位叫做Broca区，负责让我们有能力组织出符合语法的句子。语法组织能力使现实世界的交流有意义、有条理、有上下文关系。作为现实世界的模型，软件抽象在定义和公开各种契约时，也应该使它们具备同样的有意义的交流能力。

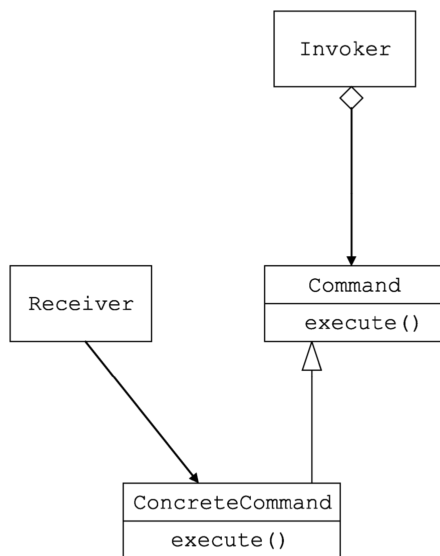
我们平常使用电脑时，操作系统会时不时下载一些安装包、更新包和新版本的系统。这些组件并不是全都由同一个人开发的，也不是全部同时开发的。但它们能够顺利地互相通信，无缝地组合在一起，并用抽象化的方式向你隐藏所有的实现差异。

以目前的软件开发生态来说，并不存在一种编程语言既能扮演兼容层，同时又满足各方面的要求。相反，我们用功能强大的运行时和中间件作为宿主，容纳多种语言、协议和分布式机制，并在它们之间实现相互通信。软件开发时，不同的组件会用不同的语言。组件代码的规模可能只有几行，也可能是成百上千行。但不管是什么样的组件，我们希望它们可以像预制单元一样组合，并且无缝地连接到其他软件基础设施构成的生态系统中。

面向对象编程可运用聚合、参数化、继承等技巧，将小的抽象发展成大的抽象。但这些技巧都有其负面影响，每一次运用都应该详加考虑。例如，在Java中使用实现继承会使类结构之间出现不必要的耦合，因而损害扩展能力。如果严格遵循设计模式所建议的最佳实践，就可以避免这类负面影响。A.3节已经举过一个设计模式的例子，我们用DI模式消除组件中多余的细节，达到撇除杂质的目的。接下来让我们看看设计模式还可以在哪些地方发挥作用。

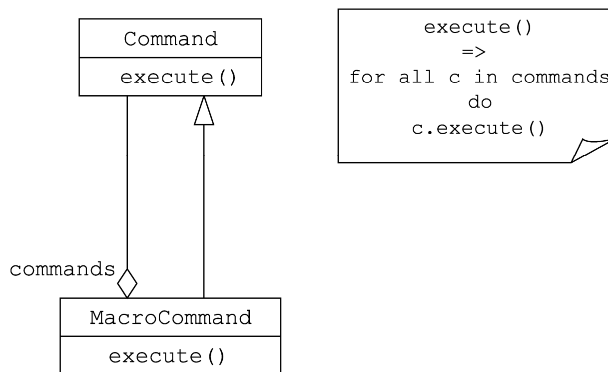
A.5.1　用设计模式满足组合性

有一种常用模式可以向客户提供对一组动作的抽象，Gamma等人称之为Command模式（参见A.6节文献[3]）。这种设计模式的目的是把对动作的请求封装成一个对象，这样就可以用不同的请求对客户作参数化，还可以对请求排队、记录请求日志，实现操作撤销和恢复功能。图A-4表现了Command模式要求的抽象结构。



图A-4　Command使调用者及接受者与具体动作解耦。因此命令对象即使脱离了当前的执行上下文，仍然可以重用

Command设计模式将调用者及接受者与将要执行的命令解耦。因此单个的命令即使脱离了当前的调用者、接受者构成的上下文，也可以单独重用。甚至可以将单个命令单元聚合起来，构成更高层次的命令。通过聚合（aggregation）的方式，命令是可组合的。图A-5表现了用聚合手段体现抽象的组合性，设计出`MacroCommand`（宏命令）的场景。



图A-5　`MacroCommand`是对命令的组合。执行`MacroCommand`等于级联地执行组成它的一系列命令

Command模式以及它的复合形式，提供了一种宏观层次的组合能力；原本独立执行特定动作的对象，变成可以编排组合的元件。但在UI的设计中，开发者需要动态增减显示组件的单个特性，此时需要比Command模式粒度更低的组合能力。抽象公开的接口是固定的，但通过该接口组合起

来的对象各有一套功能。

这种情况下应该应用Decorator模式。在这种模式下，你围绕一个核心对象设计一群包装类，也就是所谓的装饰器，装饰器的接口与核心对象相同。装饰器可在对象级别动态地装上或拆下。Decorator模式提供了子类化和实现继承以外的又一种组合途径。还有很多种结构模式和行为模式，同样被OO程序员群体用于设计拥有组合能力的类结构。

A.5.2　回归语言

随着时间推移，前面讨论过的很多设计模式已经被现代的OO语言和函数式语言纳入为语言特性。A.3节就提到过，Scala和Ruby都有支持基于mixin的继承的内置。mixin是对抽象进行组合的优秀手段，也是实现多重继承的正确途径。Scala语言中基于对象的mixin特性，就是按照Decorator模式实现的。

除了用mixin作模块化的组合，Scala的高级类型系统还提供了其他一些机制，可以提高抽象的组合性。第6章讨论用Scala作为实现语言设计复杂领域模型时，会一并详细讨论。本节重点观察编程语言日趋强大之际普遍呈现的发展趋势，即越来越多的设计模式和最佳实践被吸收成为语言实现的一部分。

1. 基于原型的OO

有一种形态的OO并不包含类的概念，只有一种抽象形式——对象——用作对实体行为的建模手段。如果想共享某些行为，只要将一个对象标记为另一个对象的父本即可。在这样的模型中，不存在任何静态的继承层次结构；实现继承在基于类的OO中表现出来的固有的缺陷都消失了。JavaScript就采用这种*基于原型*的OO模型，而我们之前讨论那种OO模型称为*基于类*的模型。我们从基于原型的OO语言的角度，观察一下它们是如何处理对象的组合问题的。

在下面的JavaScript例子里，首先定义一个`instrument`对象，作为所有其他票据的*原型式*对象。*原型式*的意思是指`instrument`对象充当基础的角色，被所有实现了同样契约的对象所共享。`fixed_income`对象是`instrument`的特化变体，在`instrument`的实现之上增加了独特的行为。在实现层面，`fixed_income`有一个指针指向它的父对象，也就是所谓的*原型*。此例中的原型是`instrument`对象。

安排以上结构的思路并不难理解。当你调用对象的某个方法时，接到调用请求的对象把这个方法（或者叫*消息*）与它自身的契约集（或者叫消息集）进行比对，如果找不到匹配的消息，那么就将消息转发给它的原型，看原型是否能响应该消息。消息逐次转发给上一级的原型，直到成功响应或者到达特殊的根对象`Object`为止。

通过原型来实现对象级别的知识共享，叫做*委托*（delegation）。委托可以在最细小的粒度层次动态地组合抽象。

```
var instrument = {
  issue: function() { //..  }
  close: function() { //..  }
  //..
```

```
}
var fixed_income = Object.beget(instrument);     将该对象的原型设为instrument
fixed_income.mature = function() { //.. }
```



要选择最合适的OO形态来对问题进行建模，而且绝不应该被语言束缚住手脚。与其为实现设计模式而编写大量的八股代码，不如把眼光放宽一点，换一种更得力的语言也许会找到更简洁的出路。

当设计模式被吸收进语言实现之后，呈现出来的结构可能不像原先那么明显，很可能已经和语言融为一体，变成一种习惯用法。Ruby中的元编程就是很好的例子。

2. 元编程（无处不在）

在Java中实现Builder模式需要大费周章，但如果通过Ruby元编程来实现，却只是一种自然的习惯用法。具体例子可以参考Jim Weirich利用Ruby的`method_missing`特性巧妙实现的一个XML标签Builder（详情可参阅http://github.com/jimweirich/builder/tree/master）。类似地，Ruby中构造新对象的惯常方式，也已经融合了DI模式。Strategy模式既可以用Ruby的模块特性直接实现，也可以利用Ruby在运行时修改类实现的能力，动态地实施。

A.5.3　副作用和组合性

上文讨论过的Command设计模式还有一些方面值得继续深入探讨。Command模式对用户动作进行了封装，开发者得以将多种动作组合成更高层次的抽象。此处的用户动作是指用户施加于对象以期产生某种结果的动作。但除了实现用户的意图外，动作还可能产生另外的副作用，例如顺带在控制台打印一些信息、向数据库发出写入请求、抛出异常，或者改变某些全局状态。

副作用的结果取决于过往历史——对银行账户执行取款动作，同时会有更新余额的副作用，更新的结果因当前余额而异。你不能贸然忽略程序语句解释执行的顺序，也不能假设编译器一定不会进行语句合并、结果缓存、缓求值之类的优化。有副作用的程序不容易被理解，更不容易分析。

1. 分离命令和查询

在面向对象编程实践中如何有效地控制副作用呢？答案依旧取决于近年发展起来的设计模式、惯用法以及最佳实践。其中一种方案叫做命令–查询分离（Command-Query Separation）模式，这是Bertrand Meyer在设计Eiffel语言期间提出的，后来得到Martin Fowler的推广（详情可参阅http://www.martinfowler.com/bliki/CommandQuerySeparation.html）。“查询”被限定为只求得结果的单纯动作，而不引起任何全局的状态变化，也不产生任何副作用。相对地，“命令”只以产生副作用为目标，通常牵涉到某种状态的变更。在该模式下，每个抽象要么属于一种查询，要么属于一种命令，但绝不可两者兼具。



副作用不可组合。运用Command-Query Separation模式区别模型中的查询和命令，如此方能有效掌握所有产生副作用的抽象。

函数式编程很少利用副作用来达到目的，即使有必要使用，也可以通过一部分语言提供的特别

标注，明确地声明具体将产生哪些副作用。函数式语言并不一定限制副作用，但Haskell会通过它的静态类型系统对副作用进行约束。在Haskell里不允许把带副作用的抽象传递给要求纯粹性的函数。

2. Haskell示例

前面我们一直用对象来实现Command模式的建模，现在不妨尝试用函数式的方式来实现。假设我们的任务是对集合中的每个元素依次施用f函数和g函数。如果按照A.5.1节的实现，我们先要把f函数和g函数分别封装成两个命令（也许封装成函数对象的形式），然后用这两个命令组合成一个宏命令。在Haskell里完成同样的任务，则要简单得多：

```
map f (map g lst)
```

map是Haskell中的一种组合子（combinator），它对列表中的所有元素分别调用由用户提供的一个函数。在上面的代码片段中，首先g函数被应用到列表lst的每个元素中，生成一个作为中间结果的列表结构。外层的map再对中间结果列表中的每个元素应用f函数，得到作为最后输出结果的列表。以上操作符合一般的逻辑，也很接近前面MacroCommand例子的思路。

前面已经说过，Haskell是一种纯函数式语言，不允许将带副作用的函数传递到要求纯粹性的地方。map刚好是一种只接受纯函数的组合子。请看map在Haskell语言中的类型定义：

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
```

map是一个函数，接受另一个函数(a->b)和一个列表[a]作为输入，通过将函数(a->b)分别应用到源列表[a]的每个元素中，生成另一个列表[b]作为结果。当我们让map调用f或g函数时，除非f和g都是没有任何副作用的纯函数，否则编译器会拒绝接受。而又因为f和g必定是纯函数，所以Haskell编译器可以将例子中的调用变换成等价的map (f.g) lst。这样一来，原先的分步调用，经过编译器的变换，变成对列表中的元素调用一个复合函数。这样的好处是用来存放中间结果的临时数据结构完全消失不见了。纯粹性的保持导致结果具有更好的组合性。

想必你对A.5.1节面向对象的Command模式实现如何处理副作用有所疑问，更想知道在Haskell的纯粹世界中怎样完成同样的事情。Haskell在语言层面实现了Command-Query Separation模式。请考虑以下函数：

```
f :: Int -> Int
g :: Int -> IO Int
```

f函数是纯函数，给它相同的输入总是能得到相同的结果。而g函数是一个带副作用的函数，从它的类型声明就可以看出来。g类型申明它返回一个动作，该动作在执行的时候会有副作用，该动作返回一个Int类型。g函数也许会从stdin或者数据库中读取某些信息，也许会修改某项可变的状态。重复调用g不一定每次都能得到相同的结果，结果取决于动作执行的历史。Haskell的类型系统明确地强调，f是一个查询，而g是一个命令。

并非所有的函数式语言都像Haskell那么纯粹。大部分函数式语言不要求带副作用的函数在签名中作任何显式声明。但这一点并不影响函数式编程相比OO编程在组合能力方面的优势。函数式编程把对纯粹性的追求渗透到日常实践之中，程序员对于那样的写法已经习惯成自然。副作用放在函数式的世界会被当做旁门左道，而在OO世界中却显得很自然。

A.5.4　组合性与并发

可组合的抽象带来的最大好处，可能是它们为并发编程做的铺垫。如果现在让你设计一个打算在多线程环境下运行的并发抽象，你大概会用基于锁的同步机制来完成设计。设计中要考虑并发性本来就不容易，而采用基于线程的执行模型，其固有的不确定性更使设计难上加难。基于锁的并发控制不可组合；在锁同步机制下单独具有原子性的操作，并不能保证组合之后仍然满足原子性。难怪计算机科学领域的研究者要竭力发明一种更好的并发控制抽象。

STM（Software Transactional Memory，软件事务内存）是已经被Haskell、Clojure等语言成功实现的一种并发控制结构。STM提供了一种类似于数据库事务的并发控制机制，可以代替锁同步机制完成程序中对共享内存的访问控制。STM的首要优点，是赋予你把原子操作组合成更大的原子操作的能力。

让我们用一段Haskell代码来具体说明。下面的片段实现了在两个银行帐户之间转账的原子操作。

```
transfer :: Account -> Account -> Int -> IO ()
transfer from to amount
  = atomically (do { credit to amount
                   ; debit from amount })
```

即使你不太熟悉Haskell语言，也不必急着去买相关书籍。这段例子非常直观，不难看出`deposit`和`withdraw`这两个原子操作在Haskell组合子`atomically`的保障下，组合成一个更大的原子操作`transfer`。

在本书第2部分介绍如何通过组合子实现高层次抽象时，组合性这个主题将会反复出现。只有当你能够组合抽象，隔离副作用时，改善抽象的表现力才是一个可以企及的目标。

A.6　参考文献

[1] Brooks, Frederick P. 1995. *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition (2nd Edition). Addison-Wesley Professional.

[2] Prasanna, Dhanji R. 2009. *Dependency Injection: Design Patterns Using Spring and Guice*. Manning Publications.

[3] Gamma E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[4] Bracha, Gilad, and William R. Cook. September 1990. *mixin-based Inheritance. Proceedings of the European Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 303-311.

[5] Kuleshov, Eugene. February 2008. *It is safer not to invent safe hash map/Java*. http://www.jroller.com/eu/entry/not_invent_safe_hash_map.

附录 B

元编程与DSL设计

元编程技术常常与DSL设计联系在一起。利用元编程技术，我们能够实现让代码生成代码。我们设计DSL时，可以利用语言的运行时设施或编译时设施生成最终的代码。但这些生成的代码可能繁复之极，又刻板生硬且难以阅读。本篇附录主要探讨一些适合于DSL设计的常用元编程技巧，假如能够利用好它们，将非常有利于提高DSL的表现力和简洁度。

B.1 DSL中的元编程

我们从2.1节得知Groovy语言具有强大的元编程能力，用它实现的DSL的表现力远远超过相应的Java实现。Groovy和Ruby这类语言允许我们动态地调整对象的运行时行为。对象在运行期间获得的各种机能使其语义具备非常高的可塑性。这些动态行为受MOP（MetaObject Protocol，元对象协议）支配，并在语言的运行时得以实现（参见B.3节文献[5]）。语言的元对象协议决定了语言各构成元素的扩展性语义。下面对编程语言的MOP做进一步的解释。

定义 元对象是一种用来操纵其他对象的行为的抽象。有些OOP语言有“元类”的概念，由其负责各种类的创建和操作。为了履行职责，元类需要保有与类有关的一切信息，如类型、接口、方法、扩展对象等。

语言的MOP决定了用该语言写成的程序的扩展性语义。程序的行为取决于MOP，程序中哪些内容在编译或运行时会被扩展也取决于MOP。

元编程可以通过编写程序来产生新的程序，也能改变已有程序的行为。在Ruby和Groovy等OO语言里，元编程能够扩展既有的对象模型，它通过添加钩子来改变既有方法甚至类的行为，也可通过运行时的内省机制置入新的方法、属性或模块。而在Lisp等语言中，则以宏作为元编程的实现手段，在编译阶段对语言进行句法层面的扩展。对比起来，Groovy、Ruby的主要元编程实现形式是运行时的元编程，而Lisp的元编程是编译时的元编程，且不会引入任何额外的运行时负担（Groovy及Ruby都可以通过库来实现对AST的显式操作，这是一种编译时元编程，但其精美程度不可与Lisp同日而语，其原因将在B.2节揭晓。）Java语言也通过标注处理机制（annotation processing）和面向切面编程（aspect-oriented programming，简称AOP）的途径实现了元编程能力，

并且完全在MOP内定义其扩展机制。

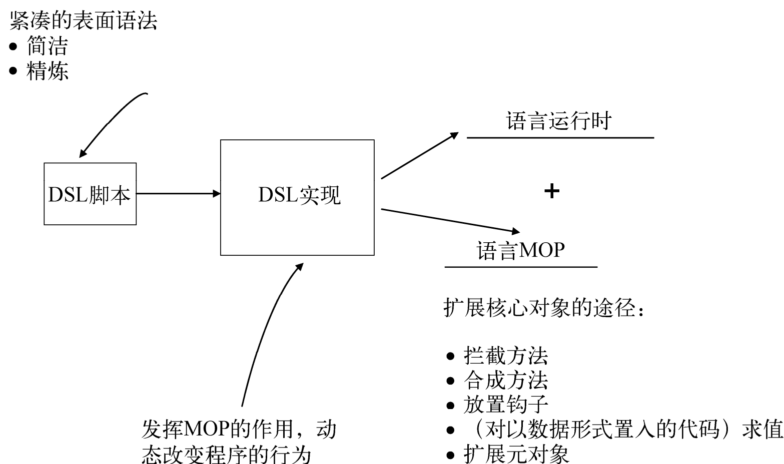
这种语言能用代码来生成代码吗？当你需要决断手头的语言是否足以胜任DSL实现时，必须要问自己这个至关重要的问题。元编程可以拓展宿主语言的语法，使之向领域用语靠拢，用在DSL设计上将发挥极大的威力。传统上依赖纯内嵌方式实现DSL语义的静态类型语言，如Haskell和OCaml，现在也分别通过Template Haskell（http://www.haskell.org/th/）和MetaOCaml（http://www.metaocaml.org/）进行扩展，并提供了类型安全的编译时元编程机制。

这一节我们分析几种时新语言的基本元编程能力及其在DSL设计中的作用。本书第二部分对这些元编程特性一一进行了深入探讨，同时提供了大量的应用实例。

B.1.1　DSL实现中的运行时元编程

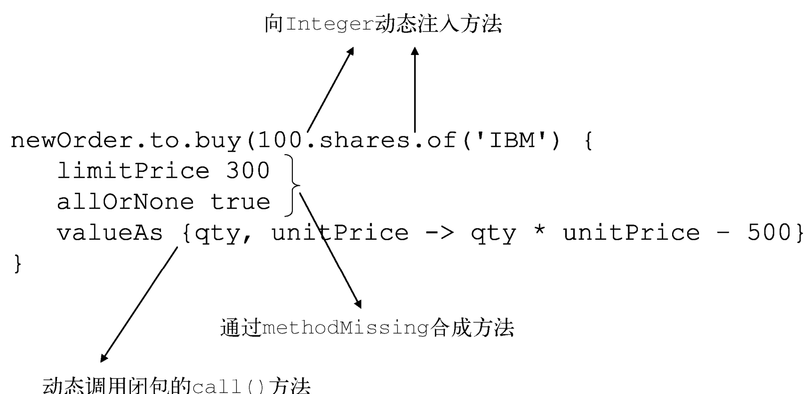
对于DSL的宿主语言来说，是否支持元编程这种语言特性为什么如此重要？因为元编程令语言拥有扩展的能力，如果我们用一种可扩展的语言来实现DSL，那么DSL就会自动地获得扩展能力。空口白话或许不好理解，我们可以用一些例子来辅助说明什么是所谓的扩展性。

图B-1显示了一种支持运行时元编程的语言的DSL执行模型。如果语言的MOP允许对各种核心语言特性进行扩展，那么我们可以在DSL实现中利用这种能力去改变和扩展一众关联对象的核心行为。这样，DSL与MOP携手将复杂的实现隐藏起来，从而使表面语法得以保持简洁。如图B-1所示，DSL脚本通过DSL实现的解译，并在核心语言运行时及语言元编程行为机制的共同作用下，最终完成其处理过程。



图B-1　语言元模型在DSL执行中承担的角色

从上面的抽象模型中，我们可以了解元编程在DSL执行过程中所扮演的角色。MOP对增强语言的表现力有着举足轻重的的作用，我们可以通过回顾第2章Groovy实现的交易指令处理DSL来说明这一点。图B-2标示了其中的关键点。



图B-2　Groovy实现的交易指令处理DSL中，元编程发挥作用的几处关键点

图B-2中每一条标注指示的位置，都在Groovy MOP所定义的元编程机制下发生了对核心语言抽象的动态修改或扩展。这些修改和扩展都隐藏于DSL实现之内，对外的契约则保持简洁精炼，并且不增加任何非本质复杂性。

B.1.2　DSL实现中的编译时元编程

我们从第2章的例子中得知，Groovy MOP通过扩展核心语言的语义来实现在运行时期间的动态程序行为。代码生成、方法合成、消息拦截这些动作，都是在程序开始执行后发生的，这意味着Groovy和Ruby的所有元对象都是语言的运行时产物。编译时元编程允许我们在编译期间构造和操纵程序。我们可以定义新的程序构造，操作编译器完成语法变换，有针对性地对应用进行优化。编译时元编程的这些能力，恰好完美地呼应了Steele提出的设想（参见B.3节文献[6]）：“*对发展的规划应该是语言设计的一个主要目标。*”的确，我们可以凭借编译时元编程，让语言的语法平滑地向着需要的方向演变。

语法宏（syntactic macro）是最普通的一种编译时元编程形式。不同语言的宏在复杂度和能力上有很大的差异，有像C语言预处理器那样简单的文本式的宏，也有像各种Lisp变体和Template Haskel、MetaOCaml等静态类型语言那样在AST层面进行操作的精巧的宏。本节我们将详细探讨宏及编译时元编程的某些能力，它们对精炼DSL的设计起到强大的推动作用。

除了宏，有些语言还提供其他依靠预处理器的编译时元编程方式，例如C++模板、面向切面编程（AOP）和标注处理机制。像Groovy和Scala等语言，还可以通过显式实现的编译器插件，以操作AST的方式获得一些元编程能力。这些编译时元编程方式我们会在下文一一谈及，其中讨论的重点是以Lisp语言家族为代表的基于宏的方案。

1. C++：模板

模板是C++语言首要的元编程机制。C++模板通过编译时期间对数据结构的操纵，获得强大的代码生成能力。模板这种编译时元编程形式在科学和数值计算方面的应用十分成功，被用来生成算法的内联版本，并运用诸如循环展开等技巧来优化算法的性能。

表达式模板（expression template）也是一种有用的C++元编程技巧（参见B.3节文献[1]），它可以有效地替代C风格的回调。回调函数不可避免地会带来函数调用的系统开销，而表达式模板把各种逻辑和代数表达式直接内联在函数体内，因而避免了相应的系统开销。C++数组处理类库Blitz++（参见B.3节文献[2]）就运用“表达式模板”的技巧来建立数组运算表达式的语法分析树，并进而产生优化的定制计算内核。将这种能够在编译时生成代码的技巧用于DSL设计时，我们可以把涉及向量、矩阵等高阶数据结构的运算代码写成下面的样子：

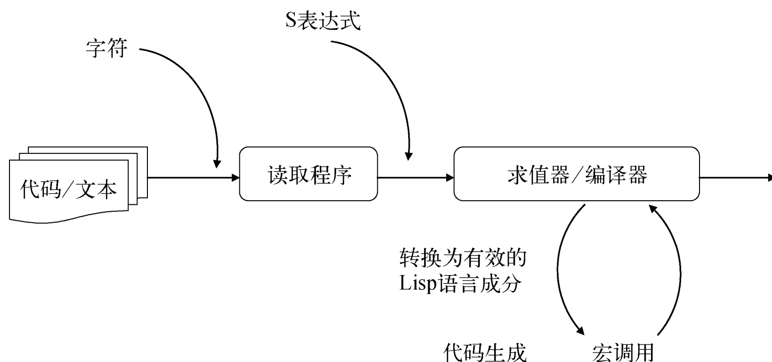
```
Vector<double> result(20), x(20), y(20), z(20);
result = (x + y) / z;
```

除了通过模板的实例化来生成代码，C++的操作符重载也是一种原始的元编程形式。作为C语言的后继者，C++也继承了C语言的宏机制，由一个位于编译器之前的预处理器来负责对宏的处理。通过宏来进行编译时元编程的，还有另外一群语言，也就是我们下一节要谈到的Lisp语言家族。

2. Lisp和Clojure：宏

Lisp的宏机制提供了最为成熟完善的编译时元编程支持。C语言的宏局限于文本替换操作，表现力匮乏；相对地，Lisp的宏可以全面调动语言的一切扩展能力。

当Lisp表达式含有宏调用时，Lisp编译器不对调用的参数进行求值，而是原样传递给宏代码。宏代码经过处理，返回一段新的Lisp语言成分来替换处理前的宏成分，然后编译器对新的表达式进行求值。对宏调用的整个转换过程完全在编译时进行，转换产生的代码完全由有效的Lisp语言成分构成，而且完全与主程序的AST合为一体。图B-3简要示意了Lisp编译时元编程机制的构成。



图B-3　Lisp语言通过宏来提供编译时元编程能力

除了语法宏之外，Common Lisp语言还有很多天生就适合元编程的特性。比如它的代码和数据有着统一的表达形式，它的递归求值模型，它的代码由表达式而非语句构成，类似这样的语言特性会给元编程带来很大的便利。

Clojure（http://www.clojure.org）是一种由Rich Hickey开发的，在JVM上的Lisp实现。Clojure也像Common Lisp一样，通过语法宏来进行元编程。由于Clojure在JVM上实现，所以它可以无障碍地与Java相集成，且具有与Java对象互操作的能力。在本篇余下的段落，我们将用Clojure代码

片段来演示Lisp语言的DSL设计之道。而且这些例子所代表的编程范式，我们也一概用Lisp来称呼。因为说到底，Clojure语言也是一种Lisp。Lisp语言本身的设计就恰好满足DSL实现对表现力的追求，其中的缘由我们会在B.2节详述。不介意的话，现在请再看一眼图B-3。图中非常简略地描绘了预编译阶段Lisp宏生成代码的过程。

现在就让我们来对这个过程作一点深入的探索，仔细地观察宏展开过程中，变换产生最终的Lisp成分，并且被编译器求值的每一个步骤。假设我们有这样一段处理客户交易指令的DSL，它的任务是根据某些条件，将交易指令提交给交易引擎：

```
(when (and (> (value order) 1000000)
          (is-premium-client? client))
 (make-trade order broker)
 (update-journal client))
```

片段中的when是一个宏，其定义如下：

```
(defmacro when [test & body]
 (list 'if test (cons 'do body)))
```

当Lisp编译器遇到宏调用时，它手头并没有可以用来对形参求值的运行时实参。编译器能看到的只有源代码。因此它将作为源代码的以下三个Lisp列表，不经求值，原样地传递给宏：

```
(and (> (value order) 1000000) (is-premium-client? client))
(make-trade order broker)
(update-journal client)
```

然后编译器以这三个列表成分为实参运行宏。形参test被绑定为列表成分(and (> (value order) 1000000) (is-premium-client? client))，而另外的(make-trade order broker)和(update-journal client)成分则被绑定到形参body。于是在宏定义体内的反引号表达式（backquote expression）作用下，宏被下面展开之后生成的新代码取而代之：

```
(if (and (> (value order) 1000000)
        (is-premium-client? client))
 (do
   (make-trade order broker)
   (update-journal client)))
```

Common Lisp的宏机制也像Groovy MOP一样，存在一个代码生成的过程，但与Groovy不同的地方在于，这个过程发生在预编译阶段。因此Lisp运行时绝对不会遇见任何元对象，在它面前出现的全部都是有效的Lisp成分。

3. Java：标注处理机制和AOP

Java也拥有一定程度的编译时元编程能力，标注处理机制（annotation processing）和面向切面编程（AOP，参见B.3节文献[4]）是它实施元编程的两个途径。Java程序里的标注会在程序构建时得到处理，而处理时生成的代码可以补充或修改原本的程序行为。

AspectJ（参见B.3节文献[3]）是Java语言的AOP扩展，它有一套数量不多但威力强大的程序控制结构，可以插入到字节码当中，从而向既有程序注入新的行为。我们可以指定程序执行路径中某些明确的点，称为*连接点*（join point），向这些点注入含有新行为定义的*通知*（advice）。连接点的集合称为*切入点*（pointcut）。切入点、通知、再加上一些相关的Java成员定义，就构成了

AspectJ的模块单元*切面*（aspect）。切面的作用是在特定的切入点上生成代码，相当于给Java增加了一套元对象协议。在Java语言下，利用切面来实现有限形态的DSL是可行的。例如Java EE框架的代表Spring（http://www.springframework.org）就通过这种途径给开发者提供了精干的领域语言，算是一个相当成功的范例。

B.2　作为DSL载体的Lisp

元编程和代码生成可以造就出色的DSL设计，这一点我们已经在2.3节有过详细的叙述。用户所期待的出色的DSL设计，除了表面语法紧凑外，还要具备充分的领域词汇表达能力。这就要求宿主语言必须具备充足的程序转换语义，这种转换可在编译层面进行，也可在运行时层面进行。

我们从2.3.1节得知，Groovy等语言利用运行时MOP生成代码，并通过诸如方法合成、方法拦截等元对象操纵手段改变程序的行为。但运行时元编程确实存在性能方面的弊病，因为变换涉及的程序结构需要通过元对象的反射和内省来实施操纵。

Lisp等语言通过语法宏来提供编译时元编程能力，这是B.1节刚刚讨论过的。正是由于语法宏的功劳，Lisp运行时完全摆脱一切元结构，执行Lisp程序时只需考虑核心语言运行时中定义的有效Lisp语法成分即可。这个特点使得Lisp元编程独树一帜，也使得语法宏成为实现Lisp DSL的根本。本节我们将更进一步剖析Lisp程序的构造，以图理解Lisp在实现DSL的方法上与其他语言的区别。

B.2.1　Lisp的特殊之处

是什么原因令Java、C++这类语言难以实施编译时元编程？要想有效率地编译时元编程，我们需要在程序的AST上执行各种变换操作。下面的内容大致说明了抽象语法树和具体语法树的含义。



在大多数语言里，我们编写的程序都会被表示成一棵CST（Concrete Syntax Tree，具体语法树）。CST真实地反映程序内容，包括代码中的空白、注释以及编写中产生的一切元信息。然后，程序依次经过扫描器、词法分析器、语法分析器的处理，生成所谓的AST（Abstract Syntax Tree，抽象语法树）。AST代表了经过一系列编译阶段，从程序代码中提取出来的、具有语法意义的实质部分。从CST到AST一般要经历变换、优化、代码生成等步骤，所有这些变换操作都主要由语言的语法分析器负责实施，变换的结果是产生AST。

大多数语言如Java或C++都用字符串来表示程序，从CST产生AST的唯一方法是通过语法分析器，而语法分析器只能分析有效的语法成分。语法分析器不是一个独立的模块，在程序的预编译阶段并没有语法分析器可供使用。（这个说法并不完全准确。现在有的语言，如第9章简略提到的Template Haskell和MetaOCaml，已经实现了基于语法宏的编译时元编程。）于是，在语法分析器缺席的情况下，这些语言如果要处理程序中的新语法或者进行预编译时的程序变换，就只能通过以下几种原始的手段：

- 像C语言那样，依靠一个预编译器来进行文本替换式的宏展开；
- 通过（Java）标注或者（C++）模板在预编译阶段选择性地做一些预处理；
- 像AspectJ在Java语言下进行AOP那样，在字节码中间插入另外的指令。

有C语言背景的读者，很可能体会过使用文本替换式的宏所带来的混乱、痛苦和小心翼翼。C语言宏的窘迫恰恰反衬出Lisp宏的高明。语法的扩展性从一开始就是Lisp的设计目标，而且相关的支持设施也已经贯穿语言的整个设计过程。当Lisp之父John McCarthy决定这种语言要能够访问其自身的抽象语法时，就注定它会成为现在的样子。

到目前为止，本篇基本都在谈论宏。它操纵AST，将新的语法转换成基本的Lisp成分。宏之所以能够成为Lisp的扩展性来源，归根结底，还是在于Lisp语言本身的设计。独特的设计哲学塑造了Lisp这种与Java和C++截然不同的语言，下文将略述其中几点。

B.2.2　代码等同于数据

在Lisp语言里，所有的程序都是一个列表结构，同时这个列表结构也是代码本身的AST。这条规则导致程序代码与数据的具有完全相同的表达形式和语法。如果再推而广之，让语言的抽象语法也遵守这条简单的设计规则，那么我们就可以像访问代码和数据一样访问抽象语法，而显然这抽象语法也是一个极为简单的列表。我们用Lisp制造的任何元程序都只要遵守这种简单的、统一的表达形式即可。

B.2.3　数据等同于代码

我们可以通过Lisp的特殊成分`quote`，毫不费力地在表示代码的语言构造中嵌入表示数据的构造。Lisp宏即为这种用法思路的代表例子。实际上，Lisp将它*数据等同于代码*的范式做了进一步的拓展，形成一种可用于编写元程序的完善的模板机制。这种机制在Common Lisp语言中称为*拟引用*（quasiquotation）。Clojure语言也具备同样的特性，由*语法引用*（syntax quote）、*解引用*（unquote）和*接合解引用*（splicing unquote）几部分构成。我们可以看下面的例子，这是Clojure语言中`defstruct`宏的定义：

```
(defmacro defstruct
 [name & keys]
 `(def ~name (create-struct ~@keys)))
```

语法引用由反引号（`）表示，其作用是指示Lisp将紧跟在反引号之后成分视为数据，效果与一般的引用相同。但是我们可以在被施加语法引用的成分内部，用解引用符号（~）指示Lisp停止对指定成分的引用，并对该成分求值。Common Lisp语言的拟引用也具有类似的作用，我们可以用它来定义数据模板，其中一部分数据是固定的，而另一些数据则是计算得出的。这一套语言特性几乎相当于在Lisp的语法里内嵌一种完整的模板子语言。

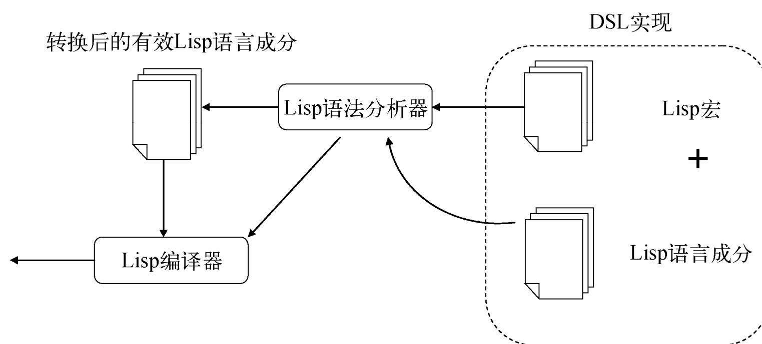
第5章详细介绍了元编程的实践，并对Lisp这种对代码和数据一视同仁的特性进行了深入探讨。如果你还没有习惯Lisp的各种编程范式，那么现在可以停下来，好好想象一下这种特性会给代码生成带来怎样的精彩和活力。

B.2.4　简单到只分析列表结构的语法分析器

Lisp是一种语法极其精简的语言。Lisp的语法分析器之所以如此简单，是因为它需要分析的就只有列表而已！无论是数据还是代码，其表达的语法都是统一的列表结构。甚至我们所关心的Lisp宏，其宏体部分也是一个列表结构。

具备强大编译时元编程能力的Lisp，是一种同像（homoiconic）的语言。这一点跟Lisp之所以具有卓越的DSL实现能力有关系吗？答案很简单：我们可以贯彻Lisp的“同像”哲学，把DSL也表达成一个列表结构，并且用宏来组织DSL中出现的重复性的构造和模式。这样设计出来的DSL不需要任何额外的语法分析器，可以将一切都交给Lisp本身的语法分析器去处理。宏可以帮助我们突破Lisp成分的形式限制，拓展出新的语法和语义，向领域用语靠拢。图B-4形象地说明了用Lisp语言作为DSL载体的基本思路。

定义　同像（homoiconic）这个煞有介事的术语，描述的是语言的一种性质。如果一种语言的程序，能够用它本身所能处理的一种数据结构来表示，我们就说这种语言是“同像”的，以Lisp为例，它用列表这种结构来统一地表示代码和数据。



图B-4　作为DSL载体的Lisp。Lisp宏被转换为有效的Lisp成分，然后送到编译器

你能够从图B-4中看出Lisp是怎样集外部DSL和内部DSL于一身的吗？一方面，DSL里面含有外部语法，也就是各种宏。宏不是有效的Lisp成分。另一方面，我们不需要使用任何外部的分析器去处理这些外部语法。列表结构串起了所有的环节，而且Lisp本身的语法分析器就是万能的DSL处理器。我们在此讨论的Lisp语言的众多特点几乎使它成为一种完美的DSL实现语言。

元编程是让我们通过编写程序来编写程序的一种技术。Lisp用它的编译时宏机制来实现元编程。B.1节是对编译时元编程的全面综述，而本节则针对Lisp这种最早具备元编程能力的语言之一，讨论了该语言下的具体实现。只有对“元”的力量有了透彻的理解，我们才能在现实的DSL实现中得心应手地运用这种范式，并领略其中的妙处。第4章和第5章准备了大量动态语言的的编译时和运行时元编程例子，所用语言包括Ruby、Groovy和Clojure。

B.3　参考文献

[1] T. Veldhuizen. Expression templates. *C++ Report*, 7 (5) pp. 26-31, June 1995.

[2] Blitz++, http://sourceforge.net/projects/blitz/ .

[3] AspectJ, http://www.eclipse.org/aspectj/.

[4] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. 1997. Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming*, pp. 220-242.

[5] Kiczales, Gregor, Jim des Rivieres, and Daniel G. Bobrow. 1991. *The Art of the Metaobject Protocol*. The MIT Press.

[6] Steele, Jr, G.L. Growing a language. *Higher-Order and Symbolic Computation* 12 (1999), pp. 221-236.

附录 C

Ruby语言的DSL相关特性

本篇附录将帮助你熟悉Ruby语言中有助于DSL开发的一些特性。请不要把本附录看作一篇细致而全面的语言综述。如果希望完整而详细地探讨Ruby语言及其语法，可以参阅C.2节所列的文献资料。

C.1 Ruby语言的DSL相关特性

Ruby是一种动态类型的OO语言，它的反射式元编程和生成式元编程能力都非常强。Ruby的对象模型允许我们通过对元模型的反射，在运行时改变对象的行为。它在运行时生成代码的元编程能力，也可以被我们用来精简DSL的表面语法。表C-1简要概括了那些令Ruby成为优秀的DSL实现语言的重要特性。

表C-1 Ruby语言特性汇总

<table>
<tr><th colspan="2">类和对象</th></tr>
<tr><td>Ruby是面向对象的语言。我们可以定义类以及类中的实例变量和方法
一个Ruby对象拥有一组实例变量，并与一个类关联
一个Ruby类是Class类的实例。它除了拥有对象所拥有的一切，还含有一组方法定义，以及指向超类的一个引用
我们在使用Ruby语言设计DSL时，用类来建模领域实体是一种惯常的做法</td><td><pre>class Account
  def initialize(no, name)
    @no = no
    @name = name
  end
  def to_s
    "Account no: #{@no} name: #{@name}"
  end
end</pre>initialize是一个特殊的方法，会在我们调用Account.new时被执行。它的作用是在对象获得初始内存分配之后，设置好对象的状态
@no和@name设置类的实例变量。变量使用之前不必事先声明</td></tr>
<tr><th colspan="2">单例（singleton）</th></tr>
<tr><td>我们可以定义只针对单一特定对象的方法，是为单例方法。
Ruby类中的方法定义其实也都是一些单例方法而已，它们是针对某个Class类的实例而定义的。Ruby的单例也称为类方法</td><td><pre>accnt = Account.new(12, "john p. ")
def accnt.do_special
  ##
end
accnt.do_special ## runs
acc = Account.new(23, "peter s. ")
acc.do_special ## 错误!</pre>do_special方法只针对accnt实例定义。若在这个单例方法内引用self，将会指向accnt实例</td></tr>
</table>

（续）

元　编　程

元编程是Ruby DSL成功的秘诀。Ruby是一种具有反射能力的语言，允许我们接触运行时的元对象并改变其行为

```
class Account
  attr_accessor :no, :name
end
```

attr_accessor是一个运用了反射式运行时元编程手法的类方法。它会为参数中输入的属性生成相应的读写访问方法。这个例子充分展示了元编程对于表面语法的精简效果，那些刻板机械的部分都被放到运行时去生成

```
class Trade < ActiveRecord::Base
  has_many :tax_fees
end
```

这个例子用到了Rails的ActiveRecord库。这里用了一个类方法来表达实体间的一对多关系，且在施加该关系的时候运用了反射式元编程

开　放　类

Ruby允许我们在运行时打开任何类，并在其中增加或修改属性、方法等

这个特性被通俗地称作猴子补丁，一般认为它是Ruby最为强大而又危险的特性之一

猴子补丁的作用范围是全局命名空间，因此我们必须谨慎地使用该特性

```
class Integer
  def shares
    ##
  end
end
```

Ruby的开放类可以用来设计一些辅助性的成分，为DSL的语法构造提供便利。例如我们可以打开Integer类，并向其中加入shares方法。这样DSL的用户就可以按照平常的说话习惯，把代码写成2 shares。这样做的缺点是所有Integer类的使用者都会被该猴子补丁所影响。请务必小心这一点

求值操作

Ruby可以在程序执行中间随时分析、执行一个字符串或者代码块。求值操作是最有力的Ruby元编程特性之一

我们可以利用Ruby的几种求值操作来设置适当的执行上下文。假如我们传递的代码块不需要在调用方法的时候明确指定执行的上下文，那么DSL的语法也会显得简洁一些

Ruby的几种求值操作分别可以设置不同的执行上下文

- class_eval——在一个类或者一个模块的上下文内求值一个字符串或代码块
- instance_eval——在一个类实例的上下文内求值一个字符串或代码块
- eval——在当前上下文内求值一个字符串或代码块

```
class Account
end
Account.class_eval do
  def open
    ##
  end
end
```

这里的上下文是Account类。class_eval将在Account类上创建一个实例方法

```
Account.instance_eval do
  def open
    ##
  end
end
```

这里的上下文是self所指的单例类。instance_eval将在Account上产生一个单例方法（或者叫做类方法）

模　　块

模块的作用是把一些相关的程序制品，如方法、类等组织在一起，方便作为一个mixin组件混入到类

模块在Ruby语言中还起到命名空间的作用

```
module Audit
  def record
    ##
  end
end
```

（续）

模　块

这个模块定义了一个新的命名空间来收纳所有与审计相关的方法。我们希望哪个类具有审计功能，就把它混入哪个类：

```
class Account
  include Audit
  ## 此处可使用record方法
end
```

代码块（block）

Ruby的代码块是一种代码的构造单元，有点类似于匿名方法，可适时经过具体化（reification）之后执行。它也像方法一样可以接受参数输入

Ruby的代码块是lambda的同义词，可以用来实现高阶函数

```
sum = 0
[1, 2, 3, 4].each do |value|
  sum += (value * value)
end
puts sum
```

竖线括起来的`|value|`即是传递给代码块的参数

Ruby数组的`each`方法接受一个代码块作为它的参数

用散列充当变长参数列表

Ruby可以很方便地实现不确定长度的方法参数列表

我们只需要用一个散列结构来充当传递的媒介，然后按照键值对的方式访问里面的参数

这种手法可以提高DSL代码的可读性，同时令Builder模式的实现变得极为简单

```
def foo(values)
  ## valued 是一个散列
end
```

函数的调用方法：

```
foo(:a => 1, :b => 2)
```

这种惯用法的应用示例：

```
class Trade
  has_many :tax_fees,
    :class_name => "TaxFee",
    :conditions => "valid_flag = 1",
    :order => "name"
end
```

鸭子类型

在Ruby语言里，我们基于类型来设计抽象，而是基于抽象所响应的消息。如果一个对象响应表示鸭子叫的`quack`消息，那么它就是一只鸭子！

这种写法在Java语言里是行不通的。Java语言要求我们给方法的参数指定明确的类型

```
class Duck
  def quack
    ##
  end
end
class DummyDuck
  def quack
    ##
  end
end
def check_if_quack(duck)
  duck.quack
end
```

不管参数中输入的实例属于`Duck`类还是`DummyDuck`类，`check_if_quack`方法都能得到正确的结果，因为两者都响应`quack`消息

C.2　参考文献

[1] Thomas, Dave, Chad Fowler, and Andy Hunt. 2009. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*, Third Edition. The Pragmatic Bookshelf.

[2] Perrotta, Paolo. 2010. *Metaprogramming Ruby: Program Like the Ruby Pros*. The Pragmatic Bookshelf.

附录 D

Scala语言的DSL相关特性

本篇附录将帮助你熟悉Scala语言中有助于DSL开发的一些特性。请不要把本附录看作一篇细致而全面的语言综述。如果希望完整而详细地探讨Scala语言及其语法，可以参阅D.2节所列的文献资料。

D.1 Scala语言的DSL相关特性

Scala是一种在JVM上运行的，兼有面向对象和函数式编程范式的语言。由于Scala与Java共享对象模型（以及很多其他方面），所以两者的互操作性十分优秀。Scala的语法简练优美，具有类型推断能力，还因为综合了OO和函数式两种范式，因而拥有十分丰富的抽象设计机制。

表D-1 Scala语言特性汇总

基于类的OOP	

Scala是一种面向对象的语言。我们可以定义含有实例变量和方法的类。除了类之外，Scala还有很多其他的表示类型的语言构造适合用于抽象设计，而且每一种都有其自身的特点和应用范围。本篇附录会尽量介绍它们
按照Scala DSL设计的一般习惯，无论类还是别的能够把若干相关功能组织成一体的语言构造，都常常被用来建模领域实体
类定义语法的详情请参阅D.2节文献[1]
	`class Account(val no: Int, val name: String) {`
	`  def balance: Int = {`
	`    // 实现`
	`  }`
	`  //..`
	`}`
	类的定义可以加上参数。在上面的代码片段中，`no`和`name`前面的`val`意味着它们都是不可变的，不允许再次赋值
	`balance`是一个方法。它没有任何参数，返回值类型为`Int`

case类	

只要类定义前面加上case字样，编译器就会生成一种附带诸多福利的抽象。这种抽象Scala称为case类。对于一个case类，编译器会自动施行以下动作
■ 转换构造器的参数为不可变的`val`。不希望被转换的参数可明确标明为`var`
■ 为该类实现`equals`、`hashCode`和`toString`方法
■ 允许使用简写形式来调用构造器。初始化该类的一个对象时，不需要写出new关键字。编译器会产生一个伴随对象，含有用来构造对象的`apply()`方法和提取构造参数的
	`abstract class Term`
	`case class Var(name: String) extends Term`
	`case class Fun(arg: String, body: Term) extends Term`
	按照这段case类定义，我们可以用`val p = Var("p")`的写法来实例化一个对象，不需要明确写出new关键字

（续）

case类	
`unapply()`方法 case类的另一个作用是参与模式匹配。它最常也最习惯被用来实现一些参与代数运算的数据类型 由于case类天然地具有各方面的不可变特征，我们在设计DSL的时候常常用它来实现不可变的值对象	
trait	

trait也是Scala表达抽象的一种手段。它和Java的接口（interface）类似，允许将具体的实现留给具体类去完成。但trait有一点和接口不一样，它可以选择性地给一部分方法提供默认的实现
trait是Scala实现mixin的机制，也提供了一条正确实现多重继承的途径
	`trait Audit {`
	`  def record_trail {`
	`    // 实现`
	`  }`
	`  def view_trail // 保持开放`
	`}`
	`class SavingsAccount extends Account`
	`with Audit {`
	`  //..`
	`}`
	trait善于设计开放的、不绑定到特定实现的、可重用的抽象。在上面的trait定义里，`view_trail`方法保持开放的状态，留待混入该trait的抽象去实现

高阶函数和闭包	

在Scala语言里，函数与其他的可作为值传递的类型完全平等，我们可以把一个函数作为参数传递给另一个函数。函数也可以返回另一个函数。函数和值的等同性赋予了Scala实施函数式编程的强大能力
高阶函数可以精简代码，且便于我们在DSL中表达正确的动词语义
闭包能让我们少写一些类和对象，多用函数式的程序构造
	`val hasLower =`
	`bookTitle.exists(_.isLowerCase)`
	`def foo(bar: (Int, Int)=>Int) {`
	`  //..`
	`}`
	第一个例子的`exists`方法通过它的参数获得一个函数来处理字符串中的每一个字符。如果用Java语言来实现同样的功能将会烦琐很多
	第二个例子演示了Scala的函数字面量（function literal）语法

模式匹配	

Scala也拥有函数式编程语言必备的模式匹配功能。我们可以匹配任意的表达式，匹配过程会在找到第一个匹配项时成功结束，即以顺序为优先（first-match-wins）的匹配规则
我们在Scala语言下施展函数式的编程手法，case类和模式匹配的组合可以说是杀手锏
Case类可以用来实现可扩展的Visitor模式
我们从第6章的例子可以体会到，模式匹配是清晰表达业务规则的利器
	`def foo(i: Int) = i match {`
	`  case 10 => //..`
	`  case 12 => //..`
	`  case _ =>`
	`}`
	这段代码使用Scala的语法简单复制了Java语言的switch/case语句。其实模式匹配还有很多别的用法
	`val obj = doStuff()`
	`var cast:Foo = obj match {`
	`  case x:Foo => x`
	`  case _ => null`
	`}`

（续）

模式匹配

Java语言下的`instanceOf`检查，按Scala的习惯一般会写成上面的样子

```
trait Account
case class Checking(no: Int) extends
Account
case class Savings(no: Int, rate:
Double) extends Account
def process(acc: Account) = acc match
{
  case Checking(no) => // 执行操作
  case Savings(no, rt) => // 执行操作
}
```

Case类可直接用于模式匹配。Case类默认实现的属性提取器（extractor）在匹配过程中起到重要作用

起模块作用的object语法

Scala的`object`语法定义了一种可执行的模块。我们使用类和trait定义好的各种抽象，通过`object`语法把它们组合为一个具体的对象实体。Java语言中与`object`语法最为接近的对应物是静态内部类

```
object RuleComponent extends Rule
with CountryLocale with Calendar {
//..
}
```

`RuleComponent`是用声明中所列抽象组合而成的一个单例对象

隐含参数

我们可以将函数的最后一个参数声明为`implicit`，从而达到调用时省略该参数的目的。编译器会在包围该函数的作用域内查找匹配的参数

```
def shout(at: String)(implicit curse:
String) {
  println("hey: " + at + " " + curse)
}
implicit val curse = "Damn! "
shout("Rob")
```

如果无法在作用域内找到匹配的参数，编译器将提示编译错误

隐式类型转换

隐式类型转换可以帮助我们在不对现有库进行任何改动的前提下，实现对该库的扩展。其原理类似于Ruby的猴子补丁，但多了词法作用域的约束

Martin Odersky把这种手法称为Pimp My Library模式（参见D.2节文献[2]）

编译器会在需要的时候自动调用隐式转换函数。我们可以利用隐式类型转换让旧的抽象适应新的API

```
class RichArray[T](value: Array[T]) {
  def append(other: Array[T])
    : Array[T] = {
    // 实现
  }
}
implicit def enrichArray[T](xs:
Array[T]) = new RichArray[T]
```

这里定义的带`implicit`关键字的`enrichArray`函数，是一个从`Array`类型到`RichArray`类型的转换函数

（续）

偏　函　数	

偏函数只对其参数所有可能取值中的一部分有定义。Scala的偏函数形式上呈现为含有一连串case语句的模式匹配代码块
习惯上我们常常把Scala actor的消息接收循环定义成偏函数
	`val onlyTrue:`
	`PartialFunction[Boolean, Int] = {`
	`  case true => 100`
	`}`
	`onlyTrue`是一个限定了参数取值范围的`PartialFunction`。它只针对`Boolean`值为`true`的情况定义。`PartialFunction` trait含有`isDefinedAt`方法，会在参数取值满足偏函数定义域的时候返回`true`。以下是两个例子：
	`scala> onlyTrue isDefinedAt(true)`
	`res1: Boolean = true`
	`scala> onlyTrue isDefinedAt(false)`
	`res2: Boolean = false`

泛型和类型参数	

Scala允许在类和方法的声明中指定类型参数。而且我们还可以对这些类型显示指定一些抽象必须满足的约束条件。对约束条件的检查将由编译器自动执行，我们无需自行编写任何验证代码
我们设计DSL时，可以善加利用Scala语言的特点，尽量把DSL的约束条件纳入其类型系统
	`class Trade[Account <:`
	`TradingAccount](account: Account) {`
	`  //..`
	`}`
	按照这里的类定义，不满足约束条件的账户将无法生成相应的`Trade`实例

D.2　参考文献

[1] Wampler, Dean, and Alex Payne. 2009. *Programming Scala: Scalability = Functional Programming + Objects*. O’Reilly Media.

[2] Odersky, Martin. Pimp My Library. *Artima Developer*. http://www.artima.com/weblogs/viewpost.jsp?thread=179766.

附录 E

Groovy语言的DSL相关特性

本篇附录将帮助你熟悉Groovy语言中有助于DSL开发的一些特性。请不要把本附录看作一篇细致而全面的语言综述。如果希望完整而详细地探讨Groovy语言及其语法，可以参阅E.2节所列的文献资料。

E.1 Groovy语言的DSL相关特性

Groovy是一种动态类型的OO语言，拥有强大的反射式元编程和生成式元编程能力。Groovy与Java语言共享对象模型，因此它们之间的互操作性十分优秀。Groovy还可以作为一种脚本语言使用。Groovy比较重要的语言特性包括可选的类型声明、运算符重载、便捷丰富的字面量语法，以及闭包等函数式抽象。表E-1简要概括了那些令Groovy成为一种优秀的DSL实现语言的重要特性。

表E-1 Groovy语言特性汇总

<table>
<tr><th colspan="2">基于类的OOP</th></tr>
<tr><td>Groovy是一种面向对象的语言。我们可以定义类以及类中的实例变量和方法。类的定义语法与Java类似，不过省略不写的可见性修饰符会被默认为public。类定义语法的详情请参阅E.2节文献[1]</td><td><pre>class Account {
  Integer balance(Date date) = {
    // 实现
  }
  //..
}</pre>上面是Groovy类的声明片段</td></tr>
<tr><th colspan="2">可选的类型声明</th></tr>
<tr><td>我们可以像使用Java语言那样静态地为运行时声明类型；也可以用def关键字来代替类型声明，从而获得像Python语言那样的动态类型效果。方法和闭包的形参甚至连def关键字都可以省略不写</td><td><pre>String str = new String("Groovy");
str = 8

def dstr = "dynamic"
dstr = 20</pre>str将被赋值String类型的the String 8
dstr将被赋值Integer类型的the Integer 20</td></tr>
</table>

（续）

属　性	

不管什么类型的字段，只要省略不写该字段的可见性修饰符，就相当于声明了一个属性	class Foo {
	  String str
	  def dyn
	}
	这是一个含有属性的类

字　符　串	
我们可以定义单行的字符串、多行的字符串，以及可以内嵌占位符的GString	def single = '这是单行字符串' def multi = """ 这是多行字符串""" def gstring = "$single 一共有 ${single.size} 个字符" 这段代码展示了Groovy支持的几种字符串
集合数据类型	

Groovy提供了各种常用的集合数据类型，如Range、List、Map，等等。它们各自都有十分简便的字面量定义语法，尤其适合用DSL脚本	// （半开）区间
	(0..<10).each { println it }
	// 各种列表操作
	[1,2,3] * 2 == [1,2,3,1,2,3]
	[1,[2,3]].flatten() == [1,2,3]
	[1,2,3].reverse() == [3,2,1]
	[1,2,3].disjoint([4,5,6]) == true
	// map定义的字面量语法
	def map = [a:0, b:1]
	上面是Groovy语言中各种集合数据类型的例子

闭　包	

闭包是一种可以在适当时机具体化（reification）之后执行的代码块。闭包内封装了一段逻辑，也封装了包围这段逻辑的作用域	def clos = { println "hello world!" }
	clos() // 结果打印输出"hello world! "
	def mult = {x, y -> println x * y}
	mult(2, 5) //结果打印输出 10
	上面是Groovy闭包的一些简单的使用片段

建造器（builder）	

建造器可以用惊人简洁的语法构造出复杂的层级数据模型。其中的秘诀是元编程	def builder = new
	  groovy.xml.MarkupBuilder(writer)
	    builder.html(){
	      head(){
	        title("Welcome"){}
	      }
	      body(){
	        p("How are you?")
	      }
	    }
	Groovy建造器的实现原理结合了元编程和闭包

元编程——ExpandoMetaClass	
ExpandoMetaClass是Groovy最重要的元编程构造之一，它允许我们按照闭包的定义语法，动态地添加方法、构造器、属性和静态方法	Integer.metaClass.twice << {delegate * 2} 这段代码向Integer类添加了一个名为twice的方法。新的方法在不受限制的作用域内对所有的线程可见

（续）

元编程——category特性	

category概念的作用与ExpandoMetaClass类似，但可以将动态注入内容的可见性限制在我们明确指定的作用域内。	`class IntegerCategory {`
	`  static Integer twice(Integer i) {`
	`    return i * 2`
	`  }`
	`}`
	`use (IntegerCategory) {`
	`  assert 4 == 2.twice()`
	`}`
	twice方法仅在use {}划定的作用域内可见

E.2　参考文献

[1] König, Dierk, Paul King, Guillaume Laforge, and Jon Skeet, 2009. *Groovy in Action*, Second Edition. Manning Early Access Program Edition. Manning Publications.

[2] Subramaniam, Venkat. 2008. *Programming Groovy: Dynamic Productivity for the Java Developer*. The Pragmatic Bookshelf.

附录 F

Clojure语言的DSL相关特性

本篇附录将帮助你熟悉Clojure语言中有助于DSL开发的一些特性。请不要把本附录看作一篇细致而全面的语言综述。如果希望完整而详细地探讨Clojure语言及其语法，可以参阅F.2节所列的文献资料。

F.1 Clojure语言的DSL相关特性

Clojure是一种植根于JVM的函数式编程语言，以通用编程语言为定位。它属于动态类型的语言，具有类型推断的特性，还可根据需要向编译器提供类型提示（type hint）以提高效率。Clojure是一种Lisp方言，直接编译成JVM字节码执行。Clojure语言具有同像（homoiconic）的特征，且内建了丰富而强大的并发控制结构。

除了表F-1所列的项目，Clojure还有很多值得了解的语言特性，包括并发和状态管理方面的特性、各种缓求值序列（lazy sequence）、序列推导（sequence-comprehension）和循环，以及众多高级数据结构。详情可以查阅F.2节文献[1]。

表F-1　Clojure语言特性汇总

函数式——围绕函数来组织DSL	

我们的整套DSL全部由函数构成
 	
函数是Clojure首要的语言构造。Clojure的函数支持高阶函数和闭包特性，也支持匿名函数
 	
Clojure虽然建立在Java之上，但它的函数式特征完全占据主导地位。尽管Clojure允许我们下降到Java层面去调用对象语义，但就语言习惯来说，Clojure是函数式的
	`(str "hello" " " "world")`
	`=> "hello world"`
	`(count [1 2 3 4 5])`
	`=> 5`
	`(+ 12 20)`
	`=> 32`
	前缀语法现在已经不会让人感到陌生
	注意：
	Clojure具有“同像（homoiconic）”的性质。函数调用操作本身也是一个列表，以函数的名字作为列表的第一个元素
	`(filter even? [1 2 3 4])`
	`=> (2 4)`
	`even?`是一个Clojure函数，其作用是当输入为偶数时返回`true`。我们在上

（续）

函数式——围绕函数来组织DSL	

	面例子中将`even?`函数作为参数之一传递给`filter`，`filter`会用它来逐一筛选输入列表的每一个元素
	`(filter #(or (zero? (mod % 3))`
	`                (zero? (mod % 5)))`
	`        [1 3 5 7 9 10 15])`
	`=> (3 5 9 10 15)`
	我们还可以把匿名函数传递给`filter`

函数式——函数定义	

我们通过`defn`宏来定义函数。如右边的例子所示，其语法非常纯粹。函数本身也是数据，只不过开头的位置添加了一个有语义意义的符号，这是Clojure（以及其他Lisp变体）最为引人注目的特点	`(defn ^String greet`
	`  "Greet your friend"`
	`  [name]`
	`  (str "hello, " name))`
	函数定义以`defn`开头。然后是文档字符串（docstring），也就是函数的说明文档。接着是放在一个vector里的参数列表。最后是函数体
	我们可以根据需要插入元数据，元数据以^前缀开头。例中的`^String`标明了函数的返回类型
	这段函数定义本身也是一个Clojure列表结构，定义的每一个构成部分都是列表中的元素

抽象设计	
Clojure方式	**传统OO方式**
■ 字段公开 ■ 对象不可变 ■ 由多路方法（multimethod）和协议（protocol）实现多态 ■ 不存在实现继承	■ 数据都作为私有成员隐藏在类中 ■ 对象是可变的 ■ 由层级化的继承关系形成多态，继承关系可以是接口继承，也可以是实现继承 ■ 允许实现继承
序　列	

Clojure语言的任何一种集合数据类型都是一个序列。我们可以一视同仁地对待所有种类的序列，通过相同的API来操作它们。另外，所有的Java集合类型都可以当做Clojure序列来对待，如右边的例子所示	`(first '(10, 20, 30))`
	`=> 10`
	`(rest [10, 20, 30])`
	`=> (20, 30)`
	`(first {:fname "rich" :lname "hickey"})`
	`=> [:fname "rich"]`
	在这段代码中：
	■ 第一个例子在一个`List`上调用`first`
	■ 第二个例子在一个`Vector`上调用`rest`
	■ 第三个例子在一个`Map`上调用`first`

序列同时也是函数	

Clojure把所有的序列类型都当成函数看待。这种观点源自序列的数学定义
Clojure的各种序列都可以从数学的角度去表述，下面是几个例子：
	`(def colors [:red :blue :green])`
	`(colors 0)`
	`=> :red`
	`(def room {:len 100 :wd 50 :ht 10})`
	`(room :len)`

（续）

<table>
<tr><th colspan="2">序列同时也是函数</th></tr>
<tr><td>■ Vector是它的位置的函数
■ Map是它的键的函数
■ Set是它的成员关系的函数</td><td><pre>=> 100
(def names #{"rich hickey" "martin odersky"
"james strachan"})
(names "rich hickey")
=> "rich hickey"
(names "dennis Ritchie")
=> nil</pre></td></tr>
<tr><th colspan="2">创建序列</th></tr>
<tr><td>Clojure提供了一系列的序列创建函数。其中很大一部分函数返回的序列是缓求值序列，因此可以用于创建无限长的序列</td><td><pre>(range 0 10 2)
=> (0 2 4 6 8)
(repeat 5 3)
=> (3 3 3 3 3)
(take 10 (iterate inc 1))
=> (1 2 3 4 5 6 7 8 9 10)</pre></td></tr>
<tr><th colspan="2">对序列进行筛选</th></tr>
<tr><td>Clojure为序列的筛选操作提供了若干组合子。我们应该优先使用这些组合子，尽量避免程序中出现显式的递归</td><td><pre>(filter even? [1 2 3 4 5 6 7 8 9])
=> [2 4 6 8]
(take 10 (filter even? (iterate inc 1)))
=> [2 4 6 8 10 12 14 16 18 20]
(split-at 5 (range 10))
=> [(0 1 2 3 4) (5 6 7 8 9)]</pre></td></tr>
<tr><th colspan="2">对序列进行变换</th></tr>
<tr><td>Clojure提供了大量对已存在序列进行变换操作的组合子。它们以一个序列作为输入，输出变换后产生另一个序列或值</td><td><pre>(map inc [1 2 3 4])
=> (2 3 4 5)
(reduce + [1 2 3 4])
=> 10</pre></td></tr>
<tr><th colspan="2">数据结构的持久性和不可变性</th></tr>
<tr><td>Clojure语言中所有的数据结构都具有不可变（immutable）和持久（persistent）的性质。也就是说，即使一个对象发生了改变，我们仍然可以访问它的所有历史版本。而且Clojure的实现方式并不需要额外占用大量的存储空间</td><td><pre>(def a [1 2 3 4])
(def b (conj a 5))
a
=> [1 2 3 4]
b
=> [1 2 3 4 5]</pre></td></tr>
<tr><th colspan="2">宏</th></tr>
<tr><td>Clojure进行DSL设计的秘诀是宏。宏是一种在宏展开阶段被展开替换为有效Clojure语言成分的程序结构。
宏是我们为DSL量身定做各种语法结构的利器</td><td><pre>(defmacro unless [expr form]
  (list 'if expr nil form))</pre>我们在这个宏里定义了一种类似于if和while的控制结构。它在使用的时候和一般的Clojure语言成分并没有什么两样</td></tr>
</table>

F.2　参考文献

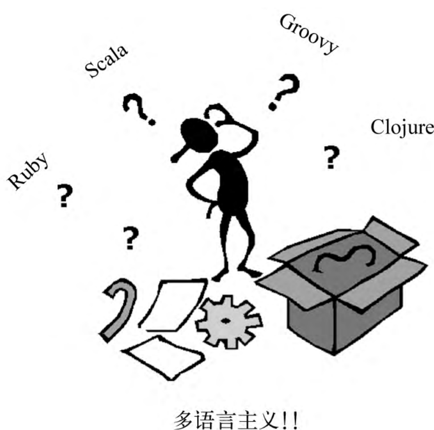
[1]　Halloway, Stuart. 2009. *Programming Clojure.* The Pragmatic Bookshelf.

附录 G



多语言开发

通读全书，你会产生一个印象：DSL不必总用一种语言来编写，我们可以根据需求来选择最适合的语言。然而当各种语言不加选择地凑在一起，互相格格不入时，语言间的摩擦又可能令我们的应用成为灾难的现场。当然，这种情形是可以避免的。那么，如何判断自己的项目是否远离了语言冲突的漩涡呢？很简单！当你真正遇到语言冲突时，肯定会像图G-1的程序员那样挠头的。



图G-1　别让自己落到这个地步

本篇附录的作用是引导读者搭建一个有序的多语言开发环境。如果我们希望在JVM上开发DSL应用，那么此时Java语言将在开发中扮演基本宿主语言的角色。应用的主体部分使用Java语言，DSL的部分则选择其他语言，从而满足目标客户对于API表现力的要求。在此有个小小的提醒——本篇附录是为初涉多语言范式下DSL开发的读者准备的，已经有过相关经验的开发者完全可以忽略。

我们会用两个例子来说明开发环境的搭建方法，例中将按照我们的设想，使用Java以外的JVM语言来开发DSL，然后将之集成到基于Java的应用主体。第一个例子针对动态类型语言Groovy，主要展示Java和Groovy语言的混合项目在现代IDE里天衣无缝的集成效果。第二个例子针对静态类型语言Scala，讲解Java和Scala混合项目的开发环境配置。

G.1 对IDE的特性要求

就JVM平台上的多语言项目来说，我们希望IDE具备以下特性。

- 支持Java与另一种JVM语言，如Scala、Groovy、Ruby或Clojure的混合项目，以及相应的项目依赖项。
- IDE所含编辑器应该具备丰富的语法功能，可以为开发者提供一定程度的协助。所谓“丰富”，指的是编辑器具备语法高亮、类型推断、鼠标文档提示、代码补全等类似功能。
- 可以在统一视图下浏览所有的项目部件，包括用不同语言编写的类型、包、视图等。
- 集成了相关语言的调试能力。

除此之外，不同的语言还可能要求一些其他的特性。静态类型语言的IDE支持一般要比动态语言的好一些，因为静态类型的程序含有较多的元信息。当前IDE的发展日新月异，众多流行的IDE都在尽力从使用者的角度去提升功能，希望开发者获得更舒适的使用感受。

G.2 搭建Java和Groovy的混合开发环境

真正从事过Java项目开发的人肯定都有使用Eclipse（http://eclipse.org）、NetBeans（http://netbeans.org）等现代IDE的经验。当我们步入多语言DSL开发领域时，自然会希望所使用的IDE能够为项目的操作和构建提供水平相当的支持。当前这方面的进展极为迅速，读者可以关注相关开发平台的更新消息。

Groovy与Java的集成关系非常融洽。我们在第3章提到过，Groovy共享了Java的对象模型，因此任何适合Java项目的IDE至少能够为Groovy项目提供水平相当的支持。不过这里面有一个隐晦的缺陷。Groovy是一种动态语言，不刻意要求指明类型，所以很多时候IDE无法得知运行时才能确定的类型信息。于是像代码补全之类的高级编辑特性，在遇到Groovy代码时就不一定能很好地发挥作用。即便有这样理论上的弱点，我们还是可以看到这个领域的持续进步。众多能够执行各式智能操作的编译器插件被发明出来，就连动态语言也不例外。

请读者按照表G-1的建议配置Java项目下的Groovy DSL开发环境。

表G-1 搭建Groovy DSL开发环境的步骤

步　骤	用　途
下载Java Development Kit（Java 5以上版本）	除了用于常规的Java开发，Groovy开发也需要Java运行时
下载NetBeans IDE（最新版本）；具体的版本兼容信息请查阅http://netbeans.org 网站上的文档	这个IDE负责管理我们的Java和Groovy项目
在NetBeans菜单里选择创建普通的Java应用	我们将要创建的Java和Groovy源文件都归属于这个应用
给项目命名，然后开始创建Java和Groovy源文件	IDE会在统一的视图下管理项目中的Java和Groovy部件。Groovy DSL脚本和Java源文件都被放置在我们设定的包结构之中。不需要任何额外的插件，Netbeans就能顺利构建这样的项目。采用第2章、第3章、第4章里讨论的任意一种方法，我们都可以轻松地在Java类里调用写好的DSL脚本

G.3　搭建Java和Scala的混合开发环境

众所周知Scala是一种拥有强大类型系统的静态类型语言。针对Scala语言的IDE支持也正在持续地发展进步，其中Eclipse、IntelliJ Idea和NetBeans已具备相当优秀的Scala代码编辑能力。

在Eclipse上添加Scala语言支持非常简单，只要安装最新版本的插件就可以了。照着下面的完整配置步骤做，就能配置好一个支持Java和Scala混合开发的Eclipse环境。

首先需要准备以下软件：

- Java Development Kit 6；
- Eclipse Classic（确切版本请查阅http://eclipse.org）。

安装好Eclipse后，就可以开始安装Scala语言插件了。插件所在的Scala IDE网站（http://www.scala-ide.org/）的主页上有一段视频，演示了安装所需的详细步骤。

Scala语言的Eclipse插件每天都在改进。插件为我们的Scala/Java混合开发工作提供了大量的功能：

- 支持Scala和Java的混合项目；
- 支持代码补全、类型推断等功能的高级编辑器；
- 增量编译；
- 调试器支持；
- 以及这里无法一一提及的、专为各种Scala和Java制品而准备的诸多功能。

G.4　常见的多语言开发IDE

表G-2列出了一些常见的、适合多语言开发的IDE。此外，还列出了每一种IDE所支持的常用语言以及相应的语言支持插件。

表G-2　适合多语言开发的IDE

IDE	支持插件
Eclipse（http://eclipse.org）	各语言的支持插件： ■ Groovy（http://groovy.codehaus.org/Eclipse+Plugin） ■ Ruby ■ Scala（http://scala-ide.org） ■ Clojure（http://code.google.com/p/counterclockwise/））
NetBeans（http://netbeans.org）	各语言的支持插件： ■ Ruby（http://netbeans.org/projects/ruby/） ■ Clojure（http://www.enclojure.org） ■ Scala（http://wiki.netbeans.org/Scala） ■ 内建支持Groovy，无需插件

（续）

IDE	支持插件
Emacs（http://www.gnu.org/software/emacs/）	Emacs支持的JVM语言很多，其中Clojure是最对它脾气的。Emacs可以说是编辑Clojure代码的首选。不过有一点需要提醒：没有用惯Emacs的开发者必须花一点时间才能适应它的各种模式（mode）。如果你打算尝试Emacs和Clojure的组合，可以先从http://www.assembla.com/wiki/show/ clojure/Getting_Star ted_with_Emacs入手
IntelliJ IDEA（Text to be displayed http://www.jetbrains.com/idea/）	各语言的支持插件： ■ Groovy（http://www.jetbrains.com/idea/features/groovy_grails.html） ■ Ruby（http://www.jetbrains.com/idea/features/ruby_rails.html） ■ Scala（http://confluence.jetbrains.net/display/SCA/Scala+Plugin+for+IntelliJ+IDEA） ■ Clojure（http://www.assembla.com/wiki/show/clojure/Getting_Started_with_Idea_and_La_Clojure）

当前IDE领域的发展十分活跃，不断增加的新特性令它们的功能越来越丰富和完善。因此我们在选择IDE的时候，最好还是先到相关的网站上做做功课再下决定。

索 引

A

B

C

D

F

G

H

J

K

L

M

N

P

Q

S

T

V

W

X

Y

Z

“本书涵盖了DSL方方面面的知识，特别可喜的是其不仅话题广泛，而且极有深度。”

——Jonas Bonér
Typesafe公司CTO

“本书全面细致，经过了精心雕琢。”

——Guillaume Laforge
SpringSource Groovy项目负责人

“本书全面介绍了DSL的定义、作用、使用时机和使用方式。”

——David Dossot
程序员、*Mule in Action*作者

“详尽、出色，而且引人入胜。”

——Federico Tomassetti
都灵理工大学助理研究员

“5种语言与丰富的实用示例，集于一书。”

——John S. Griffin
Overstock.com



DSLs in Action

领域专用语言实战

“本书很好地讲解了DSL，首先阐述了DSL的重要性，然后带我们轻松实现首个内部DSL。作者真正循序渐进地带我们领略DSL，这本书让我获益匪浅。……关于内部DSL的讲解，这是我至今见过的最为实用且完备的指南。而且，作者在讲解外部DSL的同时详细介绍了重要概念，给出了使用Xtext等的示例。”

——亚马逊读者评论

本书是第一本领域专用语言（DSL）的实战类图书，它面向真正的实践者，是一本让奋战在开发第一线的实干家备感实用的专著！

《领域专用语言实战》基于一系列JVM语言（Java、Ruby、Groovy、Scala和Clojure），分析了它们在实现DSL中的优劣势，同时也给出了丰富的示例。如果你是一位架构师，希望设计出极具表现力的API，让它既能满足领域用户的需要，又能达到高效开发的要求；如果你是一位有技术背景的特定领域用户，期待着改善与开发团队的沟通效果；如果你是一名程序员，正为如何与领域用户对接业务规则而苦恼……那么，请看这本书。

本书分三部分，以金融中介系统为例全面细致介绍从头设计、实现与使用领域专用语言的方方面面。它不仅讲解了DSL的实现手法，而且从最基本的技术入手，逐渐深入到高级技术，如元编程、解析器组合子，以及ANTLR、Xtext等开发框架。看完这本书，你将全面、深入地理解领域专用语言的定位、规范、术语，同时还可以把它作为具体开发工作的指导，学以致用，提高工作效率和质量。



MANNING

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/程序设计/DSL

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-33174-8

定价：69.00元